Ambient References: Addressing Objects in Mobile Networks

Tom Van Cutsem * Jessie Dedecker * Stijn Mostinckx †
Elisa Gonzalez Theo D'Hondt Wolfgang De Meuter

Programming Technology Lab
Vrije Universiteit Brussel
Brussels – Belgium
{tvcutsem,jededeck,smostinc,egonzale,tjdhondt,wdmeuter}@vub.ac.be

Abstract

A significant body of research in ubiquitous computing deals with *mobile networks*, i.e. networks of mobile devices interconnected by wireless communication links. Due to the very nature of such mobile networks, addressing and communicating with remote objects is significantly more difficult than in their fixed counterparts. This paper reconsiders the *remote object reference* concept – one of the most fundamental programming abstractions of distributed programming languages – in the context of mobile networks. We describe four desirable characteristics of remote references in mobile networks, show how existing remote object references fail to exhibit them, and subsequently propose *ambient references*: remote object references designed for mobile networks.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—distributed languages; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords pervasive computing, ubiquitous computing, mobile ad hoc networks, remote object references, language design

1. Introduction

The past couple of years, pervasive and ubiquitous computing have received more and more attention from academia and industry alike. Wireless communication technology and mobile computing technology have reached a sufficient level of sophistication to support the development of a new breed of applications. Such applications involve software running on mobile devices surrounded by a *mobile network*. 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 [17], thus far we observe that, at the software-engineering level, there is 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, 6]. The distributed programming languages developed to date have either been designed for high-performance computing (e.g. X10 [9]), for reliable distributed computing (e.g. Argus [15]) or for general-purpose distributed computing in fixed, stationary networks (e.g. Emerald [13], Obliq [8], E [19]). None of these languages has been explicitly designed for mobile networks. They lack the language support necessary to deal with the radically different network topology.

This paper directly focusses on distributed programming language support for mobile networks. This language support is founded on what we have previously named the *ambient-oriented programming* paradigm [10]. This novel paradigm of computing is based on the hardware phenomena fundamentally distinguishing mobile from fixed networks and advocates languages which explicitly incorporate language support for dealing with them. Within the boundaries of this paradigm, this paper reconsiders one of the most fundamental language abstractions of a distributed object-oriented programming language: the remote object reference. We show why there is a mismatch between remote object references in their current incarnation in contemporary distributed languages and the dynamically demarcated mobile networks in which they must operate.

The paper contributes to the intersection of two research areas, to wit programming language design and ubiquitous computing. Four characteristics of remote references are identified which are necessary to expressively address and communicate with objects in mobile networks. Subsequently, a family of referencing abstractions named *ambient references* is introduced which exhibit those necessary characteristics. In order to motivate the need for better referencing abstractions in mobile networks, we first discuss the impact of the mobile hardware on software in section 2. After presenting the characteristics in section 3, we introduce an actor-based computational framework to anchor our explanation of ambient references. Ambient references themselves are introduced in section 5 and are shown to exhibit the characteristics in section 6. We briefly touch upon the implementation of ambient references in section 7. Before concluding, we summarize related work and discuss limitations and future work.

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

† Author funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

2. Motivation

Based on the fundamental characteristics of mobile hardware, we discern a number of phenomena which mobile networks exhibit. Because these phenomena are so innate to the hardware from which a mobile distributed system is composed, they form a solid foundation for the desirable characteristics of remote object references explained in section 3.

There are two discriminating properties of mobile networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range. The type of device and the type of wireless communication medium can vary, leading to a diverse set of possible applications. One can imagine such devices to be physical objects "digitized" via RFID tags, embedded devices (e.g. a car's on-board computer) or simply tiny computers such as cellular phones or PDAs. All of these devices can in turn be interconnected by diverse wireless networking technology, with ranges as wide as WiFi or as limited as IrDA.

In such a hardware landscape, the most modest type of applications are so-called *collaborative applications* [14] which are based on e.g. a number of PDAs or laptops spontaneously interacting with one another (e.g. a collaborative text editor). More interesting applications can be envisaged where e.g. streets, buildings, public transportation networks and shops are all equipped with wireless networking technology, allowing for the exchange of all kinds of information between the system and proximate citizens, travellers, customers, etc. Such applications are not far-fetched, they exist or are being prototyped as we speak.

Mobile networks composed of mobile devices and wireless communication links exhibit a number of phenomena which are rare in their fixed counterparts. In previous work, we have remarked that mobile networks exhibit the following phenomena [10]:

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 reestablished. Quite often, such transient disconnections should not affect an application, allowing both parties to continue their collaboration where they left off.

Ambient Resources. In a mobile network, devices hosting services or resources spontaneously join with and disjoin from the network. As a result, in contrast to stationary networks where applications usually know where to find their resources via URLs or similar designators, applications in mobile networks have to find their required resources dynamically in the environment. Moreover, applications have to face the fact that they may be deprived of the necessary resources or services for an extended period of time. In short, resources are *ambient*: they have to be discovered on proximate devices.

Autonomous Devices. In mobile wireless networks, devices may encounter one another in locations where there is no access whatsoever to a shared infrastructure (such as a wireless base station). Even in such circumstances, two devices should be able to discover one another in order to start a useful collaboration. These observations lead to a setup where each device acts as an autonomous computing unit: a device must be capable of providing its own services to proximate devices.

In an object-oriented language, the scenarios described above can be abstractly interpreted as a set of mobile object systems embedded in a wireless ether. In such systems, remote object references form the glue between the different object systems. However, as the complexity of applications deployed on mobile networks increases, the above unavoidable hardware phenomena cannot keep on being remedied using ad hoc solutions. As we will describe next, classical remote object references break down when the object systems physically move about in unpredictable ways. In what follows, we describe necessary and desirable characteristics of remote references for mobile networks.

3. Addressing Objects in Mobile Networks

This section identifies four characteristics which remote object references should exhibit in order to adequately cope with the above hardware phenomena. We discuss the driving forces behind the characteristics and show how remote object references in contemporary distributed object-oriented languages fail to exhibit them.

3.1 Provisional Object References

In order to acquire initial remote object references to objects on remote devices, these objects have to be initially addressed via an *external description*. This description can take the form of a simple string representing an object's name (as e.g. in Java RMI) or it may be a more intensional description of a service (e.g. an interface type in JINI). In traditional, stationary, distributed systems a lookup service or name server is used to *resolve* such an external description into a remote reference. In a network composed of mobile autonomous devices, it is clear that such lookup services are too inflexible for acquiring the addresses of services. Not only do they superimpose a fixed infrastructure on the mobile network, most lookup servers make use of synchronous communication to resolve names into references. Clients query the lookup service for a name, await a response and are faced with an exception if the requested object is currently unavailable. However, in mobile networks, the chances of a requested service being temporarily unavailable are much higher than in stationary, administered networks. There is a mismatch between the synchronous request-response model and the asynchronous, event-driven nature of the physical environment.

In order to deal with the inflexibility of simple lookup services, more elaborate service discovery protocols have been devised [18]. Such protocols typically employ a peer discovery protocol based on broadcasting. In such discovery protocols, a remote reference to a service is often acquired asynchronously via publish-subscribe communication. The discovery mechanism allows clients to express their interest in a particular service and notifies them asynchronously when it becomes available on the network, usually passing along a reference to the remote service object. Asynchronous notification of discovery events has drawbacks of its own, however. It is well-known that *callback* methods used to process asynchronous replies often lack sufficient context information (i.e. the caller's state) to process the result unless this context is explicitly passed along. Also, callbacks are a source of race conditions: calls and callbacks may be processed in an interleaved manner and the callback's thread (e.g. the service discovery thread) may interfere with the calling thread (e.g. the thread that spawned a discovery request) as they usually operate in parallel on the same scope.

In short, remote object references must be acquired from an external description either synchronously, an impractical solution when services are often unavailable, or asynchronously, leading to a fragmentation of the code requesting the reference and the code using the reference. The root cause of the problem is that remote object references lack the ability to explicitly represent "objects yet to be discovered". There is no means to construct ad interim remote object references which may act as a stand-in for objects which are not available yet. Such a stand-in would allow the client to send messages to and pass around the stand-in object when the real service is not yet discovered. The discovery mechanism would then replace the stand-in by a real service when such a service would become available. Such an abstraction is very reminiscent

of the concept of a *future* [3], discussed in more detail in section 5.2.

The **volatile connections** and **ambient resources** hardware phenomena combined imply that applications will often have to refer to remote communication partners which have not been discovered yet. These phenomena thus lead us to define the ***provisionality characteristic***: the ability of remote object references to provisionally denote "objects yet to be discovered" via an external description.

3.2 Resilience to Partial Failures

Once a remote reference has been acquired, it forms a communication channel between two objects, each located on another device. However, volatile connections – omnipresent in mobile ad hoc networks – have a large impact on the behaviour of these communication channels. Disconnections usually immediately percolate into the application level by means of exceptions. The obligation to deal with potential exceptions whenever a message is sent to a remote object precludes the developer from abstracting from temporary or *transient* network disconnections, and requires clumsy `while`-loops or more advanced scheduling code to retry sending the message. Note that our argument against disconnection exceptions is no argument in favour of completely transparent distributed communication, which is impossible to attain even in fixed networks [24]. It should, however, be possible for the software developer to specify in an orthogonal manner when a disconnection may be regarded as transient and may be ignored, and when it must be dealt with as a permanent failure.

In many languages or middleware a disconnection *breaks* the remote object reference, rendering it useless. This behaviour is justified when failures are exceptional [19], but in networks where failures have a high chance of being only transient, a different mechanism is called for. In mobile networks, the expected behaviour is for the remote reference to reconnect upon re-establishing a connection. What is needed is a kind of "elastic" remote reference: when the remote device it points to moves out of range, the reference should be maintained until the device comes back in range.

The **volatile connections** hardware phenomenon, the fact that connections are often intermittent due to device mobility, leads us to postulate the ***resilience characteristic*** of remote object references: their ability to survive transient network partitions.

3.3 Transitory Addressing

Remote object references act as a designator for a remote object. A remote object reference is fundamentally different from a local object reference because it cannot address the remote object with a conventional memory address, as that object lives in a separate address space. Therefore, a remote object reference is typically implemented via a unique ID (UID) constructed from e.g. the IP or MAC address of the remote host. The remote reference only exists in one's mind's eye: it is an empty local object storing the UID, accompanied by a table at the remote host mapping this ID to a local object reference.

Unfortunately, a remote reference using a UID-based address to denote its remote object is inflexible. Remote object references are intimately coupled to the internal UID which are only valid as long as the particular remote object remains available. In mobile networks, identical services may be available on different devices. As a device roams, it is desirable to make abstraction from the specific devices hosting a service. For example, it is typically irrelevant to a user which wireless base station provides his or her laptop with internet access. Similarly, when using a cellular phone, a user is not interested in which antenna connects it to the telecom network. Moreover, as the user moves out of range of one service provider, the desired behaviour is for the application to reconnect to an equivalent provider, i.e. the "dangling" reference from client to service should rebind to an equivalent, yet not identical remote service object. UIDs are usually partly comprised of the address of a specific machine and would disallow such rebinding. Being able to seamlessly rebind remote references is a crucial step towards more self-reconfigurable mobile applications.

Generally, the programmer is forced to deal with the problem of rebinding by allocating a *new* remote object reference. The old one has become unusable and must be discarded. The fact that a new remote object reference has to be allocated for addressing the conceptually identical object opens up the possibility for unnecessary and subtle bugs if not all clients of the old remote reference consistently update their variables to contain the new remote reference.

In short, UIDs do not serve the role of a loosely-coupled, device-independent, intensional description of a remote object; their only purpose has been unique identification of a single object during the lifetime of a single application process. However, the **ambient resources** hardware phenomenon, the fact that remote services appear and disappear spontaneously, leads us to consider remote object references which use a ***transitory addressing*** scheme to designate remote objects. Relationships with remote objects may be transitory and require the remote reference to rebind to other, equivalent but not identical remote objects.

3.4 Group Communication

Mobile networks are often comprised of a good many of devices or services. A sensor network is one exemplar, but one can conceive a mobile network in a supermarket comprised of base stations, customer PDAs or wearable computers, cash registers and a myriad of RFID tags on products and shelves. In such mobile networks, it is often required to address not a single service, but rather a group or even all services of a certain type. For example, one may query for "all goods in the freezer whose expiration date is today", "all products in the customer's shopping cart", and so on.

In many distributed languages or middleware frameworks, groups of remote objects have to be represented as a collection of solitary remote object references. Unfortunately, this solution is not compositional: it precludes the programmer from treating the collection as a single remote reference that denotes an entire group of objects. This results in decreased expressiveness and leads to an increase in error-prone, duplicated boilerplate code to e.g. iterate through the collection to send a message to all members of the group. More importantly, in mobile networks groups of devices are often not statically determined, but rather form in an ad hoc manner as devices roam. For example, one is often interested in denoting a group of remote objects hosted by *proximate* devices only. It becomes very impractical to let an application manually handle such an unstable collection of proximate remote references. The application would manually and perpetually have to track the arrival and departure of nearby devices and deal with the influence of such events on the elements of the collection.

Rather than treating groups of remote objects artificially as a collection of single remote objects, collaborations in mobile networks require the plural of a remote object reference, a ***group reference*** atomically denoting an entire group of objects with a single referencing and communication abstraction.

3.5 Summary

In light of our analysis of the behaviour of objects deployed on mobile networks in section 2, we have distilled four characteristics of remote object references which are deemed necessary to properly express collaborations in mobile networks. Remote object references must be **provisional**, able to represent not yet available services. They should be **resilient** and allow communication to resume after a disconnection. Because of the constant state of flux of avail-

able services in a mobile environment, the short-lived relationships with particular service instances require a **transitory** addressing scheme, allowing a reference to be reconfigured by rebinding to equivalent services. Finally, in mobile networks comprised of large amounts of small devices, scalable programs require the ability to abstract from the parts and rather directly address and communicate with the **group** as a whole. Before we can go on to describe ambient references and how they deal with the above issues, we first need to highlight some properties of the objects which they connect.

4. Service Objects as Actors

In this section, we establish a computational framework for ambient references and some necessary terminology. This computational framework, although explained in an abstract way below, has been implemented in a concrete programming language called AmbientTalk. We postpone a description of this language until section 7. From this point on, all source code examples are pseudo code. We refrain from using AmbientTalk's syntax for didactic purposes. However, for the purposes of correctness and reproducibility, an extended version of this paper available as a technical report provides the working AmbientTalk equivalent of the relevant pseudo code examples in an appendix [22]. Before being able to describe what constitutes an ambient reference, we first describe the structure and behaviour of the *objects* they refer to.

We consider an object-oriented application model where some special objects represent certain services. Such *service objects* are special as they may be referred to by objects on other devices. Because of the inherent concurrency to which such service objects are exposed, we equip them with a model of concurrency and distribution which is heavily inspired by the actor model of computation [1] and its incarnation in stateful active objects in languages such as ABCL/1[25]. We model service objects as stateful actors. Actors are the only objects that may be remotely addressed by ambient references and communicate with one another purely asynchronously. The precise details regarding the nature of our actor model can be found in previous work [10].

Every device hosts an *actor system* which in turn hosts a set of actors. Actor systems may communicate with one another via a wireless link. Service objects (represented as actors) need a way of advertising themselves such that they can be discovered by other devices. A built-in service discovery mechanism based on publish-subscribe communication allows actors on different devices to get acquainted via an external description. This external description takes the form of a *service type*. Service types are best compared with empty Java interface types (the typical "marker" interfaces used to merely tag objects). Service types are not associated with a set of methods. Whether or not service types are aligned with interface types and hence used for static typechecking is an orthogonal design decision which is not further pursued in this paper. A service type is a subtype of one or more other service types. It denotes a set of actors which conceptually provide the same service. Service types are universal: they serve as a common ontology between all devices in the network.

An actor may declare its compliance with one or more service types, informing the actor system that the actor *provides* the services denoted by the service types. From that moment on, the actor is discoverable by other actors. In order to distinguish itself from other actors providing the same service type, an actor may accompany its service type advertisement with a property object whose public fields denote the static properties of a service. Client actors may inspect this property object to quickly filter potential communication partners based on their properties without engaging in further remote communication.

As an example, consider an instant messaging application deployed on PDAs or cellular phones where different "instant messenger" service actors may exchange text messages whenever they are in each other's proximity. Although the example may seem a bit contrived, it is a generic example of a collaborative application. Messengers may be substituted with agendas, sensors, players in a multiplayer game, products in a shop, etc. The text messages they exchange can stand for appointments, weather updates, traffic or product information, etc. Every instant messenger provides the `InstantMessenger` service and has an associated `accountid` property to uniquely identify its user:

```
servicetype InstantMessenger < Service;

method makeInstantMessenger(id) {
  return new actor {
    provide(InstantMessenger, new object{accountid=id});
    ...
  }
}
```

The service type `InstantMessenger` is declared to be a subtype of `Service`, the most general service type. Upon creation, the instant messenger actor declares that it provides this service by invoking the built-in `provide` method, passing along a property object with an `accountid` property. Objects on other devices may now refer to this service using an ambient reference, the design of which is discussed in the following section.

5. Ambient References

An ambient reference is a local representative of a remote service object. Because services are actors, ambient references are represented as actors as well. In what follows, we describe ambient references from the point of view of an application programmer. An explanation of how exactly ambient references exhibit the characteristics from section 3 is postponed until section 6.

An ambient reference is a unidirectional reference to a remote service actor created by a *client* actor interested in discovering a particular service based on an external description. An ambient reference is initialized with a required service type. For example, a client can address an instant messenger service actor by writing:

```
anInstantMessenger = ambient InstantMessenger;
```

After executing the above code, the variable `anInstantMessenger` contains an ambient reference which can bind to any available `InstantMessenger` service actor appearing in its proximity. Once an ambient reference has been constructed like this, objects can start sending it messages just as is the case with regular remote object references.

An ambient reference can be in two states: at any point in time it can be *bound* to an available remote service or it can be *unbound*. When an ambient reference is bound, we refer to the bound remote service as the *principal*. Figure 1 shows a graphical representation of an unbound ambient reference. It shows two devices, each encapsulating an actor system A and B. Their wireless communication links are represented as dotted circles which delimit their communication range. Each actor system hosts a number of actors (black circles). B hosts a service actor of a service type symbolized as a diamond (actor with embossed diamond shape). A contains an ambient reference (white circle) initialized with a service type (the diamond shape). The reference is unbound (shown dangling and dotted).

Figure 2 depicts the situation where both devices move into one another's communication range. The ambient reference is now "in

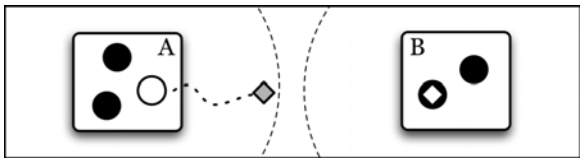

Figure 1. An unbound ambient reference

range" of a service of the required service type and gets bound (its shape fits into the provider's mould). The reference is depicted squiggly instead of rigid because its bond with the remote service may be transient: if B should move out of range, the reference becomes dangling again and may rebind to other services. This scenario is depicted in figure 3: as A and B move out of one another's communication range, the ambient reference becomes unbound. At a later point in time, A encounters a new actor system B' hosting an equivalent service (an actor of the same service type) to which its ambient reference rebinds.

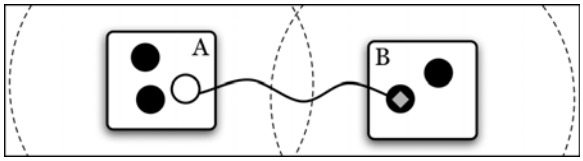

Figure 2. A bound ambient reference

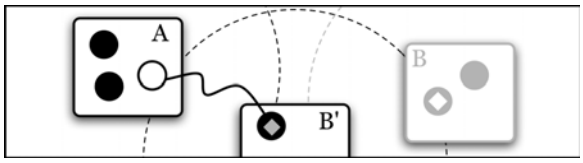

Figure 3. Ambient references may rebind.

Being a remote reference, an ambient reference is a communication channel and hence responsible for the delivery of messages sent to it to its principal. How ambient references interact with message passing is explained in more detail in section 5.2. For now, it suffices to know that message passing via ambient references is asynchronous. When a client sends a message to an ambient reference, it does not wait for the message to be forwarded by the ambient reference to its principal. Depending on the state of the ambient reference, messages are handled as follows: if the ambient reference is bound to a principal upon message reception, it forwards the message to the principal; if it is unbound upon message reception, it accumulates the message internally and forwards it whenever it gets bound in the future.

5.1 Design Dimensions in Object Designation

From our discussion in section 3 on which characteristics a remote referencing abstraction for mobile networks is to exhibit, it is clear that there is no single *right* abstraction for *all* kinds of collaborations. For example, collaborations with unknown devices encountered in a device's direct proximity are likely to be transitory and require a referencing abstraction which breaks when the service moves out of earshot and rebinds to other services as the host device moves about. On the other hand, an application running on a PDA may have a reference to a service running on e.g. the user's desktop computer at home. Arbitrarily rebinding this reference to another matching service while the user is off to work may not result in the expected behaviour. As another example, consider the group communication characteristic: some collaborations are point-to-point while others are one-to-many or many-to-many.

Rather than designing one uniform referencing abstraction, which is unable to capture all interesting forms of collaboration, we have identified three axes along which the behaviour of ambient references may vary. The remainder of this section describes each axis and the salient behaviours identified on each axis. The result of composing the three orthogonal axes gives rise to a taxonomy of ambient references. We discern three dimensions in the addressing and communication behaviour of ambient references:

The scope of binding determines which remote services an ambient reference may designate. In other words, it demarcates the set of services to which the ambient reference may bind.

The elasticity of an ambient reference directly determines its resilience with respect to volatile connections. The more elastic an ambient reference, the longer it can withstand disconnections and is able to resume its communication upon reconnection.

The cardinality of an ambient reference determines the maximum number of remote services it can represent simultaneously. This can be one, a specific few or an unknown number of services.

The differences in behaviour for each of these dimensions are discussed below.

Scope The scope of binding of an ambient reference determines to which remote services it may bind. Scoping is delimited using the service types introduced before. An ambient reference initialized with a required service type R binds to a service actor providing a service type P if and only if $P \leq R$, i.e. the provided service type must be a subtype of the required service type. Conceptually, a provider may offer a more specialized service than the one requested, but not a more general one.

The more specialized the required service type, the narrower the scope of binding of the ambient reference. Nevertheless, service types are meant to denote groups of services. It frequently happens that clients may want to distinguish between individual actors of the same service type. As described in section 4, service actors may more accurately describe their service by means of a property object. Upon constructing an ambient reference using a required service type, the scope of binding of the ambient reference may be further restricted by means of a filter query over the properties object of the service. A filter query is an arbitrary boolean expression over the fields of the property object.

As a concrete example, recall that all instant messengers are of the `InstantMessenger` service type and that each messenger has an associated property object containing the user's account ID. The code for adding a new buddy to the buddy list, given the buddy's account ID `friendID` can be written as follows:

```
buddy = ambient InstantMessenger m where
  m.accountid == friendID;
buddyList.add(buddy);
```

The above code shows how the scope of binding of the ambient reference `buddy` is restricted by the filter query `m.accountid == friendID` such that it binds only with service objects representing a specific friend. Additionally, note how ambient references act as provisional ad interim references for their remote service: the ambient reference may be readily added to the buddy list before the friend is even encountered.

In short, the scope of binding of an ambient reference consists of a service type delimiting the set of services to which the reference

may bind. If necessary, the scope can be narrowed further by providing the ambient reference with a filter query.

Elasticity The elasticity of an ambient reference directly determines its resilience with respect to volatile connections. We have chosen the term elasticity because this conjures up the mental image of references which stretch out whenever the remote actor they are pointing to moves out of communication range. If the ambient reference is elastic enough, it may survive the disconnection and allow the communication to resume. If the disconnection lasts for too long, the ambient reference snaps, like an elastic band under too much strain. We discern three types of ambient references based on elasticity:

Fragile ambient references. These ambient references break the bond with their principal from the moment the principal has disconnected. As such, the communication channel represented by these ambient references is the most susceptible to disconnection. However, remember that when an ambient reference becomes unbound it can always *rebind* later on, allowing for the communication to resume.

Elastic ambient references. These ambient references are initialized with an additional *elongation period*. This is a timeout period which specifies *how long* a disconnection may last before the ambient reference breaks the bond with its principal. Figuratively speaking, the higher the elongation period, the "further" a principal's device may wander from the ambient reference's device without breaking the bond. If a disconnection outlasts the elongation period, the reference reverts to unbound status, similar to a fragile ambient reference. The most important difference between elastic and fragile ambient references is that the former will not immediately rebind to another service when it loses contact with its principal.

Sturdy ambient references. Sturdy ambient references are ambient references which never break the bond with their principal upon disconnection. Hence, they come closest of all to the standard notion of a remote object reference. The communication channel defined by a sturdy ambient reference is most resilient to disconnections, although it pays the price of decreased flexibility (it cannot rebind to other principals). A sturdy reference may be initialized unbound. The sturdy reference then binds to the first available principal and retains this bond indefinitely.

It is clear that fragile and sturdy references can be subsumed under elastic references. An elastic reference covers the entire spectrum between fragile and sturdy references, degenerating fragile references to those with a zero elongation period and sturdy references to those with an infinite elongation period.

Depending on the kind of collaboration, different values for the elasticity of an ambient reference are appropriate. Fragile ambient references, for example, are ideal for client-service interactions that do not require session information, as it does not matter which exact service is communicated with. Another useful application of fragile ambient references is their use in encapsulating replicated services. A fragile ambient reference may be declared with a sufficiently narrow scope of binding such that it only denotes services which are each other's replica. Hence, it does not matter which service is communicated with, assuming that the replicas are e.g. interconnected via infrastructure to synchronize regularly. When a client needs the guarantee that subsequent message sends via an ambient reference are delivered to the *same* service actor, a sturdy reference is a more suitable referencing abstraction.

Cardinality The cardinality of an ambient reference determines how many remote services it can denote simultaneously. Remaining consistent with the terminology introduced by M2MI [14], we distinguish three cases:

Ambient Unireferences A unireference denotes *at most one* remote actor at a time. This is the kind of ambient reference we have implicitly assumed until now and most closely corresponds to a regular remote object reference.

Ambient Multireferences A multireference denotes *at most n* remote actors at a time, where n is the multireference's cardinality. It forms a useful group abstraction mechanism when the members or the size of the group are known upfront.

Ambient Omnireferences An omnireference denotes *all* remote actors in a given scope of binding which are available for communication. It is a flexible communication mechanism to discover an unknown number of services and to broadcast information into the surrounding environment.

In order to correctly capture group communication, ambient multi- and omnireferences are introduced. The following code excerpt declares a fragile ambient omnireference (denoted by an asterisk suffix) to address all proximate instant messengers and a fragile ambient multireference (denoted by an array suffix), addressing at most 10 *distinct* instant messengers. The multireference is *not* an array of 10 unireferences.

```
allMessengers = ambient* InstantMessenger;
tenMessengers = ambient[10] InstantMessenger;
```

Multi- and omnireferences are not simply collections of ambient unireferences. Firstly, that would miss the point of group communication abstractions as it would require explicitly managing a collection of references, rather than one group reference. Second, two separate unireferences may bind to the *same* remote service object. Hence, if the above multireference *would* have been an array of 10 unireferences, there would have been no guarantee that each unireference in the array denoted a distinct service object.

Ambient multi- and omnireferences represent a set of remote services, the *principal set*. A principal cannot occur in the set more than once. Messages sent to an ambient multireference are *multicast* to all remote services in the set. Figure 4 illustrates how an omnireference at A conceptually binds with all services of the same type available in the network.

Similar to the elasticity dimension, the cardinality dimension can be regarded as a continuum of multireferences, unireferences being multireferences with a cardinality of $n = 1$ and omnireferences being multireferences with a cardinality of $n = \infty$.

Summary Table 1 gives an overview of the different possible ambient references which can be constructed by taking the cross-product of the three described dimensions. It also shows which ambient references are parameterized by what property. The scope of binding is orthogonal to the other two dimensions. Therefore, only the combinations of elasticity and cardinality are listed (parameterized with the scope s, i.e. a service type and optional filter query). Elastic references are parameterized with an elongation period e expressed in milliseconds. Sturdy references are denoted with an

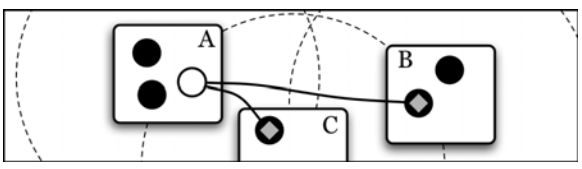

Figure 4. Ambient Omnireferences

exclamation mark to stress that their bond is fixed. Multireferences are parameterized with their cardinality n, reusing typical array syntax. Omnireferences are denoted with an asterisk to highlight their unbounded cardinality. Each entry in the table denotes an expression which, when evaluated, returns a new ambient reference of the indicated kind.

Table 1. Taxonomy of Ambient Reference Expressions

Scope of binding (s)			
Elasticity → Cardinality ↓	Fragile	Elastic `(e)`	Sturdy `!`
Uni-	`ambient s`	`ambient(e) s`	`ambient! s`
Multi- `[n]`	`ambient[n] s`	`ambient(e)[n] s`	`ambient![n] s`
Omni- `*`	`ambient* s`	`ambient(e)* s`	`ambient!* s`

5.2 Message Passing Semantics

The previous section has primarily discussed ambient references as service designators: which and how many services the reference binds to and how long this binding remains intact after disconnection. This section focuses on ambient references as a communication channel to the services they represent.

As mentioned in section 4 service objects are actors and send one another messages asynchronously. As ambient references are local stand-ins for remote service objects, communication with them is asynchronous as well. The advantage of using asynchronous, non-blocking message passing is that it decouples the client and the service in time [17]: a message can be sent to a service even when it is not online (i.e. when its representative ambient reference is unbound) at the time the message is sent. This is made possible by the ability of asynchronous message passing to decouple the act of sending a message from the act of transmitting that message. As such, messages that cannot be transmitted immediately are buffered by the ambient reference and will be transmitted later, if a connection becomes available.

We now consider how message passing is influenced by the three design dimensions. As shown below, the message passing semantics of ambient references is independent of the scope of binding and the elasticity of the ambient reference. On the other hand, the cardinality of an ambient reference has a large impact on message passing. We first detail the semantics for unireferences and gradually note the differences for increasing cardinalities.

Unireferences An ambient unireference is either bound or unbound. Messages sent to it are never lost, regardless of the state of the unireference. If it is bound, the message is forwarded to the principal. If it is unbound, messages are buffered until it becomes bound. The scope of binding and elasticity only influence the (re-)binding behaviour of an ambient reference directly, thereby influencing the message forwarding behaviour only implicitly. The following example shows how a nearby instant messenger may be queried for its user's nickname.

```
anInstantMessenger = ambient InstantMessenger;
nameFuture = anInstantMessenger#getNickname();
```

The # operator denotes an asynchronous message send. An asynchronous message send to an ambient unireference always immediately returns a *future*, which is a placeholder for the real return value. Once the real value is computed, it "replaces" the future object; the future is said to be *resolved* with the value. Futures or promises are a frequently recurring abstraction in concurrent languages (e.g. in Multilisp [12], ABCL [25] and Argus [16]). They reconcile asynchronous message sends with return values without having to resort to clumsy callback methods. Space limitations preclude us from going into more detail on the nature of futures in the AmbientTalk language. What is important to note is that futures are also actors and will not make a process block when it tries to use an unresolved future. This design is detailed elsewhere [10] and is based on that of non-blocking promises in the programming language E [19].

Multireferences A multireference has a cardinality n which is the maximum number of principals it may bind to. Other than a unireference, a multireference can either be bound, unbound or *partially* bound (i.e. when only $k < n$ principals are available). This requires a generalisation of the message passing semantics employed by unireferences. When a message is sent to a partially bound multireference, there are three possible semantics to consider. The message may be sent to all k bound principals and then discarded, the message may be stored until the multireference becomes entirely bound, or the message may be sent to all k bound principals and stored for the $n-k$ unbound principal slots. Ambient multireferences employ the third semantics because it enables messages to be sent independently to each principal at the moment it is encountered in the mobile network. On the one hand, discarding a message right away is wasteful as the chances of a principal being disconnected are high. On the other hand, waiting for all principal slots of the multireference to be bound is wasteful as the chances of all principals being connected at the same time are low. Moreover, the third semantics is the correct generalization of the semantics of unireferences, i.e. the semantics of a "multireference" with cardinality $n = 1$ coincides with that of a unireference.

Messages sent to multireferences are multicast to all principals. As a consequence, the message is duplicated and may result in multiple replies. Message sends to multireferences return *multifutures*, which are futures that may be resolved multiple times. Space considerations preclude us from going into details, which can be found in a companion paper [21]. Sturdy multireferences come closest to standard group communication abstractions: they encapsulate a fixed set and simply multicast received messages to this set. The multireference enables an asynchronous multicast whose return values may conveniently be collected via multifutures and which buffers messages for those group members not connected at message-sending time.

Omnireferences An omnireference differs from both uni- and multireferences in that it is *always* partially bound. An omnireference represents the set of *all* available services in its scope of binding. An omnireference is never completely unbound: an empty principal set is a valid set. Neither is it ever fully bound: there is no upper bound on the size of its principal set. This has important repercussions on the message passing semantics: it is clear that the message passing semantics of uni- and multireferences cannot be upheld, as this would require to somehow store a message for an infinite number of *potential* principals that may become available in the future. If the message is stored only once and duplicated lazily as new principals join the principal set, the omnireference would have to remember which messages have already been forwarded to what principal because principals may join and disjoin from the network (and hence from the principal set) an arbitrary number of times.

Ambient omnireferences employ a much simpler message passing semantics. When a message is sent to an omnireference, it is always multicast to all principals bound at *that* moment. If the principal set is empty, any message the omnireference receives is lost. Figuratively speaking, the multireference shouts the message, with the risk of no service being close enough to hear it. Message sends to omnireferences return multifutures with no upper bound on the number of expected replies.

Sustained Message Sends With respect to message sending, clients of uni- and multireferences can abstract from the state of the reference (i.e. whether it is bound or unbound) because messages are properly buffered. This is no longer the case for omnireferences. As explained above, an omnireference acts as black hole for messages when it is "empty". A typical programming idiom to deal with this fact is to send a message repeatedly at regular intervals, increasing the chances that it will eventually be received by an interested party. This idiom expresses the intent to regularly broadcast information to nearby devices. It is so inherently associated with the usage of omnireferences that the intent should be more directly expressible.

Ambient omnireferences may be sent *sustained* messages. These are messages annotated with a *decay period*, specifying how long the omnireference should store the sustained message. Upon reception of a sustained message, the message is multicast to the current principal set *and to any service joining the principal set within the decay period.*

As an example, consider the following fragile ambient omnireference hosted by a device which is part of the infrastructure of e.g. a railway station or an airport which broadcasts a timetable into the environment for interested passengers to query from their PDA. In order to reach a maximum number of passengers, each refreshed timetable is sent as a sustained message (using the `@` notation) with a decay period of 30 seconds:

```
passengers = ambient* Passenger;
passengers#announce(timetable)@30sec;
```

This sustained message is not continually broadcast during 30 seconds. Rather, the message is multicast once to the current principal set and then buffered by the omnireference for the duration of the decay period, such that it will be forwarded to services discovered later on. Ambient omnireferences *do not* guarantee that a sustained message is delivered to each principal only once. When services leave the principal set due to a disconnection and reconnect within the decay period, they may receive the same sustained message multiple times. When duplicate reception of a message is an issue, messages must be parameterized with e.g. sequence numbers to identify duplicates.

Message sends to omnireferences which are not sustained can be thought of as "ephemeral" messages having a decay period of 0 seconds. Message sends may declare an infinite decay period (via an `@forever` syntax), which effectively allows the expression of a message send targeting "all services of a given type that will ever be encountered in the future".

The problem dealt with by sustained message sends (i.e. decreasing the risk that messages are lost to empty omnireferences) is not directly dealt with by the elasticity dimension of ambient references. For example, a sturdy omnireference is a reference that bonds with all services of a certain type it ever encounters and which does not break these bonds upon disconnection (it "memorizes" who it has already encountered). When sending messages to such a sturdy variant, one may communicate with all services encountered *in the past*, but without sustained messages one would still lack the ability to communicate with services that will bind *in the future*.

5.3 Summary

We have introduced ambient references and have focussed on its two roles as a referencing abstraction: how they designate and bind to remote services and how they behave as a communication channel for messages. Ambient references are no single but rather an entire family of referencing abstractions. The salient differences between these abstractions stem from two properties: the reference's *elasticity*, the resilience of its bond to disconnections and its *cardinality*, the maximum number of remote services it denotes. A third property, the scope of binding, determines which remote services are eligible principals for an ambient reference. Ambient references feature asynchronous message passing with return values (via futures or multifutures). Messages are properly buffered by uni- and multireferences when they are unbound. Omnireferences employ a broadcasting semantics, but introduce sustained messages to address the loss of messages received while they are unbound.

This section has described ambient references from the application programmer's point of view. The following section reconsiders ambient references with respect to the three characteristics outlined in section 3. The implementation of ambient references is scrutinized in section 7.

6. Discussion

Section 3 has brought to light four necessary characteristics of remote object reference abstractions in mobile networks. As explained in section 5.1, different collaborations require different kinds of remote addressing abstractions. We discuss which ambient references exhibit or lack which characteristics, making each member of the ambient reference family suitable for a different kind of interaction.

6.1 Ambient References are Provisional

Section 3.1 addressed the need for provisional remote object references. Because required services are often unavailable, a remote object reference should be able to address services which have not yet been discovered. In doing so, the reference can abstract from the temporary unavailability of the service and the application can use the remote reference as if the service were already available. Once the ambient reference is constructed with a required service type, it can readily be used by clients as if it were a service of the desired service type (cfr. the instant messenger example in section 5.1). In the case of uni- and multireferences, the client may safely abstract from the fact that the reference may be either bound or unbound. As previously discussed, messages received by the reference while unbound are properly stored and forwarded when a service becomes available.

In the same way that futures allow one to abstract from the return value of an asynchronous message send (i.e. the result may or may not yet have been computed), ambient references allow one to abstract from the status of an asynchronous discovery request (i.e. a suitable service has or has not yet been found). Messages sent to an unbound ambient uni- or multireference are optimistically scheduled computations which are eventually triggered when a suitable service is discovered. Ambient references "objectify" services to be discovered, entirely similar to how futures "objectify" return values to be computed. Hence, ambient references are the equivalent of the well-known futures language abstraction in the context of service discovery. As such, they bring about the same advantages: they do not require an application to be artificially fragmented into callback methods to process asynchronous discovery events and allow a client to directly use a service (the return value of an asynchronous discovery request) in the same scope where it was asked for.

6.2 Resilience and Elasticity

In section 3.2, we argued for resilient remote object references: references which do not always align disconnections with exceptions or failures in order to be able to abstract over temporary disconnections.

The prime factor influencing an ambient reference's resilience is its elasticity. Using this parameter, the resilience of a remote reference to disconnections can be fine-tuned to the application's

needs. As previously remarked, there is no single "right" way of dealing with failures: some references ought to break immediately such that they can rebind to other equally useful services, other references must remain sturdy in the face of disconnections because their referent must not change. Although fragile and elastic ambient references may break upon disconnection with their principal, this does not render them useless. Uni- and multireferences revert to an unbound state and any undelivered messages are properly buffered. Although non-sturdy ambient references do not guarantee that they will rebind to the *same* principal, they are still a useful communication channel which resumes its message flow upon reconnection.

6.3 Transitory Relationships

Section 3.3 stressed the importance of a transitory addressing scheme for remote references. Such a scheme promotes reconfigurable references that may rebind to equivalent remote objects by decoupling references from device-dependent object identities. The necessity for such addressing is based on the flux of the devices in mobile networks: similar services may be available on a multitude of hosts at different locations, services are updated without global administration, devices join and leave the network unannounced, etc.

There are two factors which determine an ambient reference's transitory nature. The first factor is its scope of binding, which is delimited using a service type and an optional filter query. Service types allow clients to abstract from a service's address (its UID) similar to how URLs abstract from IP addresses, variables abstract from memory addresses, file names abstract from files etc. The second factor is the ambient reference's elasticity. The principal of elastic (and fragile) ambient references may change over time, as long as it remains within the scope of binding, enabling transitory relationships. Sturdy ambient references, once they are bound, lose their transitory addressing ability, guaranteeing a stable communication channel to one particular service at the cost of becoming as brittle as UID-based referencing mechanisms.

6.4 Group Communication

The cardinality design dimension of ambient references directly addresses the need for expressively engaging in group communication. Whereas multireferences allow for a more conventional representation of a set of remote services, ambient omnireferences provide a radically different messaging semantics. Omnireferences allow for direct interaction with *all* services within their scope of binding. Fragile ambient multi- and omnireferences form an ideal addressing mechanism to denote "clouds" of services whose boundaries are vague and change constantly due to device mobility. Such "clouds" are impossible to construct via an enumeration of individual remote references, but they can be denoted intensionally via omnireferences. Whereas sturdy multireferences represent a logical link with services whose bond is immune to the physical changes in the network, fragile multi- or omnireferences represent a physical link which breaks and binds in unison with changes in the network. When the connectivity of principals of an omnireference is determined by the wireless communication range of the host devices, ambient omnireferences form an ideal abstraction for "shouting" information to proximate devices, such as the PDAs of interested passersby in the railway station example.

7. Implementation

Ambient references have been implemented in the actor-based distributed programming language AmbientTalk [10], which is specifically designed for writing applications deployed on mobile networks. AmbientTalk has been implemented as an interpreter written in Java and J2ME. The language is conceived as an exploratory research vehicle to validate our language design experiments. It is a small kernel language, supporting a minimum of operations. Language extensions may be introduced via a metaobject protocol.

Because AmbientTalk was conceived for mobile networks, it has a built-in service discovery mechanism based on publish-subscribe communication. Actors may advertise themselves via service types, as explained in section 4. The AmbientTalk kernel has no notion of ambient references; in the kernel language, actors have to discover one another via a standard subscription mechanism. In the absence of ambient references, actors register their interest in particular service types and are informed by the kernel asynchronously via callback methods when a relevant service actor appears or disappears in the network. Ambient references have been implemented reflectively via the MOP on top of this more low-level event-based discovery system.

An ambient reference is an actor that serves the role of a *proxy* to a remote service actor. It intercepts each message sent to it (reminiscent of the way Smalltalk objects may intercept their messages via `doesNotUnderstand:`) and forwards them to its principal(s). The principal set of an ambient reference consists of a plain collection of regular UID-based remote actor references. An ambient reference is initialized with a service type and regulates the complexities of the service discovery callbacks on behalf of its client.

One aspect of the implementation which is of particular interest is the decomposition of the behaviour of an ambient reference actor in a collection of modular mixins. We have used mixin-based inheritance [5] to separate the behaviour specific to each of the three dimensions of ambient references (scope of binding, cardinality and elasticity) into separate entities. As such, the entries in table 1 are mere surface syntax for the creation of actors whose behaviour is composed out of the different mixins. The fact that the three design dimensions can be cleanly factored out into separate mixins strongly indicates their orthogonality.

Space limitations preclude us from going into more detail on the implementation of the mixins. A comprehensive overview of the design of ambient references can be found in an extended version of this paper made available as a technical report [22]. Moreover, an explanation of the full implementation details including the AmbientTalk reflective source code can be found in a companion technical report [21].

8. Related Work

We now describe different addressing and communication abstractions of computational models and languages, their applicability to mobile networks and how they differ from ambient references.

M2MI The design of ambient references has been inspired by the notion of a *handle* in the many-to-many invocations (M2MI) paradigm [14]. M2MI is a paradigm for building collaborative systems deployed on wireless proximal ad hoc networks. M2MI handles use Java interfaces just as we use service types to denote remote objects in a loosely coupled fashion.

Although M2MI has influenced the design of ambient references, there are some important differences. First, M2MI handles offer no delivery guarantees: if a message is sent to an object which is not in communication range at that time, the message is lost. There is no notion of sustained message sends as introduced by ambient omnireferences to change this delivery policy. A second difference between M2MI handles and ambient references is that asynchronous messages sent to M2MI handles do not return a value: all methods of a handle's associated interface must have a `void` return type and cannot throw exceptions. M2MI's handles are not provisional or resilient: although they may represent as yet undiscovered objects, any messages sent to this undiscovered object are lost.

In short, M2MI's handlers are a suitable remote referencing abstraction for mobile networks, but they are situated at a lower level of abstraction. As a consequence, they require the programmer to focus attention on dealing with discovery, delivery and message ordering, return values and disconnections.

Actors In the actor model of computation [1], 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. When regarded as a "remote actor reference", a mail address is neither provisional nor transitory (a mail address represents a unique, existing actor) but it is resilient to disconnections. Although its resilience makes actor-based systems perform well in open, loosely-coupled distributed systems, a mail address cannot be rebound to refer to another actor.

E The E language [19] is designed for writing secure peer-to-peer distributed programs in open networks. Interestingly, E does not differentiate between local and remote objects. Rather, it differentiates between different kinds of object references. *Near* references may only point to local objects, while references to remote objects must be so-called *eventual* references. Near references may carry synchronous method invocations, while eventual references only carry asynchronous message sends. Such asynchronous message sends immediately return promises (which are similar to futures).

E's eventual references, although providing a communication channel geared towards mobile networks, do not feature any of the characteristics exhibited by ambient references. They are not provisional but rather always point to live remote objects. They are not resilient: disconnections are treated as exceptions and once a remote reference is broken, it cannot be mended. The references are not based on a transitory addressing mechanism and cannot be rebound. This design of remote references was intentional and enforces application designs where the restoration of communication links is separated from the use of the communication links. In E, devices can reestablish contact based on a special kind of object reference named a *sturdy reference*. A sturdy reference in E can be regarded as a persistent, resilient designator for a remote object.

A sturdy reference in E is, however, not a remote object reference the way a sturdy reference in AmbientTalk is. Rather, it is a *generator* for new eventual references pointing to the remote object it designates. E's sturdy references were not designed for use in mobile networks but rather for regaining connectivity to a specific object after network partitions in traditional stationary networks. As such, they are not provisional, do not use transitory addressing or cater to group communication.

Tuple Spaces Tuple spaces as originally introduced in the coordination language Linda [11] have received renewed interest by researchers in the field of mobile computing. Adaptations of tuple spaces for mobile computing distribute the tuple space across several devices. Linda in mobile environments (Lime) [20] is one such adaptation of Linda for mobile networks.

Because tuple spaces use a very process-oriented (as opposed to object-oriented) approach to distributed computing, there is no notion of a remote object reference. However, the tuple space, regarded as a communication channel does exhibit the characteristics shown to be beneficial to mobile computing in section 3. Regarding provisionality, tuples can be placed in a tuple space well ahead before another process is available to read it. With respect to resilience, tuples are stored in the tuple space until they are read. Hence, they survive network disconnections. With respect to transitory addressing, tuples are addressable based on their semantic content which is device-independent and persistent. Group communication can be expressed by adding multiple tuples at once to the tuple space. On the downside, whereas remote references provide a private communication channel between a client and a service, tuple space-based communication is necessarily global to the entire space, which may lead to unexpected interactions between concurrently communicating processes.

ActorSpace The inability of mail addresses to represent as yet undiscovered actors have been addressed in the ActorSpace model [7]. 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 handed over via message passing. On the other hand, this disallows actors to get acquainted with other actors in a loosely-coupled, time- and space-independent manner, as is the case in Linda via tuple spaces or with ambient references using service types.

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 is parameterized by a receiver mail address and a message, is replaced by two new primitives: a `send` and a `broadcast` primitive where the receiver of the message is denoted by a pattern rather than a mail address. A message send whose receiver is a pattern, e.g. `send("InstantMessenger", "getNickname")`, 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, while the `broadcast` primitive delivers it to all matching actors. This makes patterns a provisional, potentially resilient and transitory addressing mechanism.

The semantics of the `send` and `broadcast` primitives resembles the message sending behaviour of ambient uni- and multireferences. However, in the ActorSpace model, a broadcast message is suspended until a receiver is available. Ambient references introduce a more general and flexible notion of sustained message sends. Furthermore, there is no direct analogue for multireferences nor for elastic or sturdy references in the ActorSpace model.

Jini Sun Microsystem's Jini architecture for network-centric computing [23, 2] is a Java-oriented development platform for service-oriented computing. 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. Java interface types are used as a common ontology between the devices, similar to the service types of ambient references. Clients may download the advertised proxy of a remote service and may interact with the remote service through the proxy. Although Jini's architecture is also applicable to pure ad hoc networks, its lookup service architecture works best in mobile networks with a shared infrastructure.

Jini is primarily a framework for bringing clients and services together. 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. Using the proxy technique, it is possible to construct proxy references which e.g. correctly buffer requests thereby allowing for resilience in the face of network partitions and which may internally use a transitory addressing scheme to contact their home service in a loosely-coupled manner. Hence, Jini's *architecture* is flexible enough to accomodate ambient references. However, to the best of our knowledge, Jini does not directly offer any advanced remote "service" references. By default, the proxies advertised by services communicate synchronously with their service over point-to-point protocols such as JRMP or JERI.

9. Limitations and Future Work

A number of open issues which are not adequately dealt with by ambient references are discussed below.

Service Discovery We are aware of the limited expressive power of the discovery mechanism of ambient references based on service types. The mechanism, however, is lightweight and sufficient for supporting the applications it has been designed for. We consider the use of more advanced discovery mechanisms orthogonal to the notion of ambient references. Obviously, the discovery mechanism of ambient references is part of a language design exercise; it is not meant to supplant existing standardised discovery mechanisms.

Distributed Garbage Collection Distributed garbage collection (DGC) requires a set of hosts to cooperatively clean up unreferenced garbage objects by informing one another when e.g. a remote reference has become of no use to them. In open and especially mobile networks where relationships between devices are short-lived, such cooperative DGC 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 [23].

We have described ambient references as a unidirectional reference from clients to services. A service is oblivious to any ambient references pointing to it and has no direct means of communicating with and controlling connected clients. We are looking into the incorporation of leasing into ambient references. The amalgam would be an ambient reference which, upon binding with a remote service, establishes a contract with its service under which conditions the connection between them remains valid. For example, both could agree on a lease duration and the ambient reference is then responsible for renewing the lease in time. The important difference with regular ambient references is that such references explicitly involve the remote service itself in the binding process, such that the service can be given the opportunity to e.g. react to disconnected clients.

10. Conclusion

Applications deployed on mobile networks require language constructs that abstract from the complex hardware phenomena while remaining translucent enough to deal with the inescapable issues of distributed computing. When objects are distributed over mobile devices connected by an unadministered volatile network, it is no longer trivial to discover and communicate with remote parties. Object references should be augmented with additional machinery to remain aware of the hardware constellation surrounding their device. We have named such references *ambient references*.

Ambient references are an object-oriented language abstraction that exhibit four distinct characteristics which prove essential to properly address and communicate with remote parties in mobile networks. They are provisional meaning that they can denote services based on an external description that are not yet available. They are resilient in the face of transient network disconnections. They may be rebound to equivalent yet distinct service objects on different devices via a device-independent transitory addressing scheme. Finally, as mobile networks may be populated by a multitude of small devices, it is important to make abstraction of each individual object and to address and communicate with groups of objects directly. Ambient multi- and omnireferences cater to such interactions.

Rather than designing one kind of ambient reference, we have designed a family of ambient reference kinds, the behaviour of which may vary according to three different axes. The *scope of binding* of an ambient reference demarcates the set of service objects to which the reference may bind. The *elasticity* of an ambient reference determines the resilience of its bond with a remote service with respect to disconnections. Finally, the *cardinality* of an ambient reference determines how many services it can denote simultaneously. Each combination along these design axes gives rise to a kind of reference that is suitable for a particular type of collaboration in a mobile network. These axes are clearly distinguishable not only in the design of the ambient references, but also in their implementation as a composition of independent mixin objects in AmbientTalk. Although further research is necessary to turn our proposal into scalable engineering, the orthogonality of these mixin objects strongly indicates that our analysis adequately unravels the design space of object referencing for dynamically demarcated mobile networks.

References

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

[2] ARNOLD, K. The jini architecture: Dynamic services in a flexible network. In *36th Annual Conference on Design Automation (DAC'99)* (1999), pp. 157–162.

[3] BAKER JR., H. G., AND HEWITT, C. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages* (1977), vol. 8 of *ACM Sigplan Notices*, pp. 55–59.

[4] BAL, H. E., STEINER, J. G., AND TANENBAUM, A. S. Programming languages for distributed computing systems. *ACM Comput. Surv. 21*, 3 (1989), 261–322.

[5] BRACHA, G., AND COOK, W. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming* (Ottawa, Canada, 1990), N. Meyrowitz, Ed., ACM Press, pp. 303–311.

[6] BRIOT, J.-P., GUERRAOUI, R., AND LOHR, K.-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30*, 3 (1998), 291–329.

[7] CALLSEN, C. J., AND AGHA, G. Open heterogeneous computing in ActorSpace. *Journal of Parallel and Distributed Computing 21*, 3 (1994), 289–300.

[8] CARDELLI, L. A Language with Distributed Scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, pp. 286–297.

[9] CHARLES, P., GROTHOFF, C., SARASWAT, V., DONAWA, C., KIELSTRA, A., EBCIOGLU, K., VON PRAUN, C., AND SARKAR, V. 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, 2005), ACM Press, pp. 519–538.

[10] DEDECKER, J., VAN CUTSEM, T., MOSTINCKX, S., D'HONDT, T., AND DE MEUTER, W. Ambient-oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)* (2006), D. Thomas, Ed., Lecture Notes in Computer Science, Springer, pp. 230–254. To Appear.

[11] GELERNTER, D. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (Jan 1985), 80–112.

[12] HALSTEAD, JR., R. H. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[13] JUL, E., LEVY, H., HUTCHINSON, N., AND BLACK, A. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems 6*, 1 (February 1988), 109–133.

[14] KAMINSKY, A., AND 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, 2002), ACM Press, pp. 72–73.

[15] LISKOV, B. Distributed programming in Argus. *Communications Of The ACM 31*, 3 (1988), 300–312.

[16] LISKOV, B., AND 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* (1988), ACM Press, pp. 260–267.

[17] MASCOLO, C., CAPRA, L., AND EMMERICH, W. Mobile Computing Middleware. In *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[18] MCGRATH, R. E. Discovery and its discontents: Discovery protocols for ubiquitous computing. Tech. Rep. UIUCDCS-R-99-2132, Department of Computer Science University of Illinois Urbana-Champaign, 2000.

[19] MILLER, M., TRIBBLE, E. D., AND SHAPIRO, J. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing* (April 2005), R. D. Nicola and D. Sangiorgi, Eds., vol. 3705 of *LNCS*, Springer, pp. 195–229.

[20] MURPHY, A., PICCO, G., AND ROMAN, G.-C. Lime: A middleware for physical and logical mobility. In *Proceedings of the The 21st International Conference on Distributed Computing Systems* (2001), IEEE Computer Society, pp. 524–536.

[21] VAN CUTSEM, T. A Modular Mixin-based Implementation of Ambient References. Tech. Rep. VUB-PROG-TR-06-07, Vrije Universiteit Brussel, 2006.

[22] VAN CUTSEM, T., DEDECKER, J., MOSTINCKX, S., GONZALEZ, E., D'HONDT, T., AND DE MEUTER, W. Ambient References: Addressing Objects in Mobile Networks. Tech. Rep. VUB-PROG-TR-06-10, Vrije Universiteit Brussel, 2006. Available online: `ftp://prog.vub.ac.be/tech_report/2006/`.

[23] WALDO, J. The Jini Architecture for Network-centric Computing. *Commun. ACM 42*, 7 (1999), 76–82.

[24] WALDO, J., WYANT, G., WOLLRATH, A., AND KENDALL, S. C. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet* (1996), Springer-Verlag, pp. 49–64.

[25] YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1986), ACM Press, pp. 258–268.