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 – 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.

At the software-engineering level, we observe that thus far, no general stable, robust and standard ubiquitous computing platform has emerged. Moreover, although there has been a lot of active research with respect to mobile computing middleware [22], we see little innovation in the field of programming language research. Although distributed programming languages are rare, they form a suitable development tool for encapsulating many of the complex issues engendered by distribution [6, 7]. The distributed programming languages developed to date have either been designed for high-performance computing (e.g. X10 [10]), for reliable distributed computing (e.g. Argus [20] and Aeolus [32]) or for general-purpose distributed computing in fixed, stationary networks (e.g. Emerald [16], Obliq [9], E [25]). 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 [12]. 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* are 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 a small actor-

*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)

based language which we have used to explore ambient references. Ambient references themselves are introduced in section 5 and are shown to exhibit the characteristics in section 6. We illustrate the implementation of ambient references on top of the actor language in section 7. Before concluding, we summarize related work and discuss limitations and future work.

2. MOTIVATION

Based on the fundamental characteristics of mobile hardware, we distill a number of phenomena which mobile networks exhibit. These phenomena form the direct basis for a characterisation of the desired functionality of remote object references to be used for software deployed in mobile networks in section 3. Because these phenomena are so innate to the hardware from which a mobile distributed system is composed, they form a solid foundation for the characteristics of remote object references explained later on.

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 envisaged applications. Devices might be as small as coins, embedded in material objects such as wrist watches, door handles, lamp posts, cars, etc. They may even be as lightweight as sensor nodes or they may be material objects "digitized" via an RFID tag[1]. Devices may also be as "heavyweight" as a cellular phone, a PDA or a car's on-board computer. All of these devices can in turn be interconnected by a diverse range of 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* [17] which are based on e.g. a number of PDAs or laptops spontaneously interacting with one another. Typical applications are distributed whiteboards, collaborative text editors, instant messengers and file sharing services, etc. Slightly more futuristic applications can be envisaged where e.g. airports, railway stations and bus stops are equipped with wireless base stations, allowing a passenger to obtain customized information about his itinerary. Products in shops can be tagged with digital information, accessible by customers such that they can query clothes for preferred sizes or food for the amount of calories it contains. In logistics, applications abound as products can be tagged on assembly lines, routed to the correct stock, carrier and store. Buildings become pervaded by wireless sensor networks which measure the structure's strength or which can be used to monitor air pollution. Cars communicate with one another and with intelligent roads to avoid accidents and traffic jams. These 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 [12]:

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

Ambient Resources. In a mobile network, devices spontaneously join with and disjoin from the network. The same holds for the services or resources which they host. 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, we say that 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, it is imperative that the two devices can discover one another in order to start a useful collaboration. Relying on a mobile device to act as infrastructure (e.g. as a name server) is undesirable as this device may move out of range without warning [17]. 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. Devices should not be forced to resort to a priori known, centralized name servers.

As the complexity of applications deployed on mobile networks increases, the above unavoidable phenomena cannot keep on being remedied using ad hoc solutions. Instead, they require more principled software development tools specifically designed to deal with the above phenomena. For some classes of applications – such as wireless sensor networks – such domain-specific development tools are emerging, as can be witnessed from the success of TinyOS [18] and its accompanying programming language NesC [13].

If the above scenarios are modelled and implemented via an object-oriented language, they 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, classical remote object references break down when these object systems physically move about in unpredictable ways. In the following section, 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

[1]Such tags can be regarded as tiny computers with an extremely small memory, able to respond to read and write requests.

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 or an XML advertisement in JXTA). 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 such as e.g. Java RMI's registry make use of synchronous communication to resolve names into references. A client queries a name server for a name, awaits the response, and either receives a remote object reference or is faced with an exception when the requested object is currently unavailable. However, in mobile networks, the chances of a requested remote service being temporarily unavailable are much higher than in stationary, administered networks. As the client cannot monitor changes in the lookup service, it would be forced to periodically poll the name server for the service's presence.

In order to deal with the inflexibility of simple lookup services, more elaborate service discovery protocols have been devised [23]. Such protocols typically employ a peer discovery protocol based on broadcasting. For example, JINI uses such an approach to allow clients and services to spontaneously join an unadministered network [30]. 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 to process the result. At the time the callback method is invoked, the notified object has to reconstruct the state and the scope it was in when it performed the request that gave rise to this asynchronous notification. Often, this problem can be mitigated by registering a distinct callback object (an event listener) per calling context. Even then, a manual transfer of the computational context to the callback object may be required because it does not have access to all variables available in the object that spawned it. Also, one has to beware of subtle concurrency issues such as race conditions: the discovery mechanism's thread invoking the callback operates on the same scope and in parallel with the original thread that spawned the discovery request.

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* [5], discussed in more detail in section 4.2.

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. 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. For example, consider an instant messaging application where one instant messenger tries to send a text message to a remote instant messenger:

```
try {
  instantMessenger.accept(userTextMessage);
} catch (RemoteException e) {
  deal with the exception
}
```

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 [31]. 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 [25], 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 typically uses a unique ID (UID) which may be constructed from e.g. a hash of the remote object's address, the IP or MAC address of the remote host, the time at which the object was created, etc. The remote

reference only exists in one's mind's eye: a remote object reference is implemented as an empty local object storing the UID, requiring the remote host to store an export table mapping the UID 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. Additionally, UIDs usually do not persist across crashes of the host device or across version updates of the remote service: either the remote reference becomes dangling forever, or it remains bound to an obsolete object.

Because both the UID and the export table are generally inaccessible implementation details, 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", "all cars driving next to my car", "the PDAs of all participants of the meeting", ...

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 one is often interested in denoting a group of *proximate* objects (more precisely: 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 certain communication patterns in mobile networks.

Provisional References. Volatile connections and ambient resources require remote object references to be integrated with the service discovery mechanism of the language in order to construct provisional remote references using an external description of a remote object. Such provisional references are an ad interim representation of the remote object which may not have been discovered yet.

Resilient References. Volatile connections are often transient in mobile networks. Remote object references should therefore not simply break when their underlying connection gets disrupted. They should be able to rebind upon reconnection and allow communication along the reference to resume.

Transitory References. Ambient resources imply that the available services in the environment are in a constant state of flux. Quite often, relationships with particular instances of a service are transitory such that remote references should be able to rebind to other service instances. Such reconfigurable remote references must be decoupled from the low-level object identity of the objects they bond with. Moreover, this decoupling allows such references to become persistent in the face of hardware crashes: they may rebind to the service when its host device recovers.

Group References. Mobile networks may be comprised of large amounts of small devices. In such a hardware setting, collaborations involve group communication. Groups will usually not be statically determined, but rather form in an ad hoc manner as devices roam. In order for programs to scale, an application programmer must be able to abstract from the parts and rather directly address and communicate with the group as a whole.

In the next section the AmbientTalk kernel language is described, serving as a computational framework for ambient references.

4. THE AMBIENTTALK KERNEL LANGUAGE

Before describing ambient references, we establish the computational framework in which they have been conceived along with some necessary terminology. Ambient references have been implemented in the actor-based ambient-oriented programming language AmbientTalk [12], a language designed specifically for writing applications deployed on mobile networks. AmbientTalk has been implemented as an interpreter written in Java. An interpreter exists for the J2ME platform, such that AmbientTalk also runs on PDAs and smartphones. Currently, two AmbientTalk interpreters communicate with one another via the standard TCP and UDP protocols, which may be carried both over wired ethernet and over wireless fidelity (WiFi) networks. The language is primarily meant as an exploratory research vehicle to validate our language design experiments. It is conceived as a small kernel language, supporting a minimum of operations. Language extensions may then be introduced via a metaobject protocol. A discussion of the MOP is postponed to section 7. Ambient references are reflectively implemented via the MOP. The kernel itself consists of a sequential prototype-based language and an actor-based concurrency and distribution layer.

Every device hosts at least one *actor system* and such actor systems may communicate with one another via a wireless link. An actor system is an abstract representation of a virtual machine process and is said to *host* a set of actors. Our model of concurrency and distribution is heavily inspired by the actor model of computation [1] and its incarnation in stateful active objects in languages such as the ABCL language family [34, 33]. The AmbientTalk kernel discriminates between passive objects and actors or active objects (in what follows, we use the term actor and active object interchangeably). An active object has an *inbox* or incoming message queue, containing unprocessed messages and an *outbox* or outgoing message queue, containing sent but not yet transmitted messages. An actor encapsulates its own thread of execution which is an eternal "event loop", dequeueing the next message from its inbox and processing the corresponding method. Internal concurrency within an actor is prohibited to preclude race conditions on its state. Furthermore, active objects are the only objects which can be remotely addressed. It follows that they are both the unit of concurrency and distribution. A passive object is always owned by exactly one active object. If it would be passed on to another actor, a copy is passed instead to uphold this restriction. The details regarding this double-layered object model can be found in previous work [12].

Actors communicate with one another asynchronously. When an actor `a` sends a message to another actor `b`, the interpreter places the message in `a`'s outbox, addressed to `b`. From that point on, the kernel takes care of delivering the message to `b`. AmbientTalk does not explicitly introduce remote actor references as proxies in the language. Instead, remote actors are represented by their *mail address*, as in the original actor model. If a message is sent to a remote actor's mail address, the kernel transfers the message to that actor's inbox whenever a connection with its host device is available. The kernel induces a partial order on message transmission: it may deliver messages in any order, but two or more messages from one sender to the same recipient actor are delivered to that recipient in sending order. The kernel acknowledges a successful delivery by removing the delivered message from the sender's outbox.

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, and for those readers familiar with AmbientTalk's syntax, we have provided the AmbientTalk equivalent of the relevant pseudo code examples in an appendix.

4.1 Service Discovery in AmbientTalk

Because AmbientTalk was conceived for mobile networks, it has a built-in service discovery mechanism. Based on a publish-subscribe mechanism, this mechanism 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). 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. 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.

An actor may declare its compliance with one or more service types, informing the kernel 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 attributes represent the service's static properties. The property object is piggybacked onto the list of provided service types sent in reply to a discovery request by a remote device. This allows remote actors to quickly filter a potential communication partner based on its properties without engaging in further remote communication. An actor may declare that it *requires* one or more actors compliant with a certain service type. Once it does, the kernel informs the actor every time a requested service actor (a service *provider*) becomes available (*joins*) or disconnects (*disjoins*).

The publish-subscribe mechanism is implemented by the interpreter as follows. Each interpreter periodically broadcasts its presence on the network. When two unacquainted interpreters "hear" one another, they engage in a protocol where they exchange the service types required by their local actors. Subsequently, each sends the other a list of mail addresses (and property objects) denoting service providers of the requested type. Local actors are then joined with remote actors. When either interpreter disconnects, both interpreters notify their joined actors that they have disjoined: their provider actor is currently unavailable. Further details pertaining the discovery mechanism such as the manner in which actors are notified of join and disjoin events are given in section 7.

As an example, consider an instant messaging application deployed on PDAs or cellular phones where different "instant messenger" service actors may exchange text messages or files 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, etc. The text messages they exchange can stand for appointments, weather updates, traffic information, etc. Every instant messenger provides the `InstantMessenger` service:

```
servicetype InstantMessenger < Service;

method makeInstantMessenger(nickname) {
```

```
  return new actor {
    provide(InstantMessenger);
    ...
  }
}
```

The service type `InstantMessenger` is declared to be a subtype of `Service`, the most general service type (cf. the use of `Object` in Java). Upon creation, the instant messenger actor declares that it provides this service.

4.2 Asynchronous Communication

As mentioned above, actors may send one another messages asynchronously. Although the kernel provides basic support only for simple one-way message sends without return value, AmbientTalk has been reflectively extended with more suitable message passing semantics, which is described here. Consider the scenario where one instant messenger asks another instant messenger for its user's nickname:

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

The # operator denotes an asynchronous message send. AmbientTalk adopts asynchronous, non-blocking message passing because it decouples sender and receiver in time: a message can be sent to a receiver even when it is not online (connected) at the time the message is sent [22]. This is made possible by decoupling message sending from message delivery: messages which cannot be transmitted immediately are stored in the outgoing message queue of the sending actor. Asynchronous message passing is the only message passing style in which a sender can abstract from the state of the connection with the receiver: any synchronous message passing style would either force an object to wait until the message has been delivered or force it to deal with disconnection via an exception. Using synchronous message sending is impractical as it would force actors to wait for remote actors that may be disconnected for arbitrary amounts of time to reconnect, in order for the message to be transmitted.

AmbientTalk employs the following parameter passing semantics for asynchronous sends: passive objects are passed by copy (to uphold the ownership restriction, as noted above), actors are passed by mail address. Asynchronous message sends are not easily reconcilable with return values. It requires the use of either "callback" methods or a program written in continuation-passing style in order to process results. Such programming idioms clutter the code which is why we adopt the use of *futures* or *promises*, a frequently recurring abstraction in concurrent and distributed languages (e.g. in Multilisp [15], ABCL [34] and Argus [21]). An asynchronous message send always immediately returns a future object, 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.

Most languages, including the ones listed above, make a process block on an unresolved promise or future (either implicitly by using its value in an expression or explicitly via e.g. a `touch` or `claim` operator). One notable exception is the language E, which disallows waiting for a promise to be resolved. Instead, E provides a `when`-construct which registers a closure, parameterized with the determined value, with the promise. The promise schedules this closure for execution when it has been resolved with a value. AmbientTalk adopts this `when`-construct because it allows one to deal with asynchronous replies in a completely non-blocking, event-driven yet readable manner. In the instant messenger example above, if the invoker of `getNickname` wants access to the nickname, it can do so as follows:

```
when (nameFuture) lambda(name) {
  println(name + " is online.");
}
```

The `lambda` keyword denotes the construction of a lexically scoped closure whose body is executed asynchronously at a later point in time. Code directly following the `when` statement is guaranteed to be executed before the body of the closure.

4.3 Why mail addresses are not sufficient

AmbientTalk's distribution model directly inherits from the actor model the notion of a mail address to represent remote actors. We discuss why mail addresses do not yet exhibit the characteristics discussed in section 3. Mail addresses are not provisional. A mail address is always associated with a particular actor hosted on a local or remote actor system. It is not possible to construct a mail address from an external description (e.g. a service type) out of the blue. Service types *describe* remote actors, but are themselves *not* actor addresses, e.g. it is impossible to address an available instant messenger service by writing:

```
InstantMessenger#getNickname();
```

In order to properly express this intent, the service type has to be *resolved* into a mail address first via the discovery mechanism. This mechanism uses asynchronous callbacks to inform an actor that a provider has been found which, as described in section 3.1, separates the scope requiring the service from the scope having access to the service, leading to fragmented code.

AmbientTalk actors' mail addresses are resilient to the disconnections engendered by volatile connections. This makes them a suitable referencing abstraction for mobile networks: actors can send one another messages while they are offline and be confident that the message is transmitted whenever the connection between their actor systems is restored. Of course, the catch is that the connection may never be restored. The actor model guarantees eventual delivery, but because the guarantee is unbounded in time, it is not a very pragmatic one. In other words, a mail address is perhaps too resilient to failures: it *never* breaks, precluding the reference to rebind to other actors. Mail addresses have the same limitations as traditional UID-based references in this respect, so they do not facilitate transitory conversations. Finally, a mail address represents a single actor, so it cannot be directly used for group communication.

Because of the shortcomings of mail addresses, we have introduced a more advanced referencing abstraction, or rather a suite of them, called *ambient references*. They are introduced in the following section.

5. AMBIENT REFERENCES

An ambient reference is a local representative of a remote service. Because services are modelled as actors, ambient references are technically also implemented as an actors. 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 7.

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. 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).

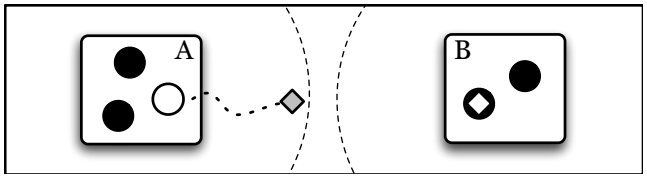

Figure 1: An unbound ambient reference

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

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. Because an ambient reference is an actor, messages sent to it are processed asynchronously. 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

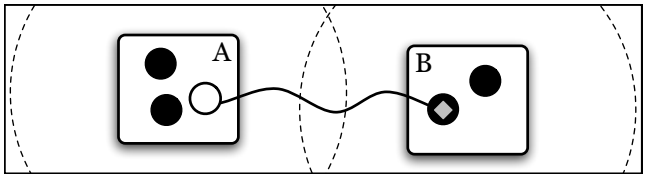

Figure 2: A bound ambient reference

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 stores the message internally and forwards it whenever it gets bound in the future. Messages are guaranteed to be forwarded to a principal in the same order as they were accumulated by the ambient reference.

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.

These observations have lead us to scrutinize the different aspects of the ambient referencing abstraction. 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 4.1, 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.

As a concrete example, recall that all instant messengers are of the `InstantMessenger` service type. In order to discriminate between different messengers, each messenger may be attributed with a user account id. Instant messengers attach this `accountid` attribute to their provided service type as follows:

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

The property object acts as a struct whose `variable=value` fields denote the properties of the service. A filter query over this property object is an arbitrary boolean expression over the fields of the property object. If the filter query accesses an attribute which the property object does not list, the query automatically fails. As an example, consider the instant messenger service whose buddy list is stored as a list of account ids. One instant messenger can then establish a communication channel with a particular buddy by creating the following ambient reference (provided `buddyId` denotes the account id of a buddy in the buddy list):

```
ambient InstantMessenger m where m.accountid == buddyId;
```

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 current 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 or that of a mail address. 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.

Although we have identified three different useful behaviours regarding elasticity, 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.

Figure 3 depicts the ability of fragile and elastic ambient references to rebind to similar services. The fragile ambient reference hosted by actor system A was bound to the service actor hosted by actor system B. 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 within its scope of binding. The ambient reference rebinds to the new service.

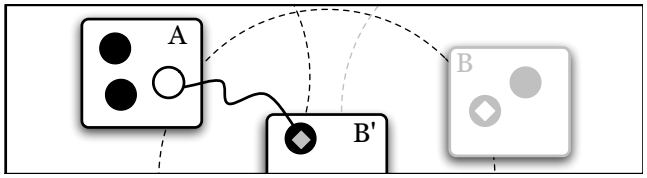

Figure 3: Elastic ambient references may rebind.

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. Sturdy references most closely resemble mail addresses, but remain provisional, i.e. they still allow a client to declare a stable communication channel to a remote service based on an external description.

Cardinality

The cardinality of an ambient reference determines how many remote services it can denote simultaneously. Remaining consistent with the terminology introduced by M2MI [17], 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 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.

Multi- and omnireferences cannot simply be represented as a collection of ambient unireferences. Consider trying to create a group of 10 `InstantMessenger` actors based on a collection of ambient unireferences:

```
group = new Vector(10);
for i = 1 to 10 do
  group.add(ambient InstantMessenger);
```

This code overlooks the fact that distinct ambient unireferences may bind to the *same* remote service object. Even if 10 distinct `InstantMessenger` actors were available at the time the loop is executed, the resulting collection may represent an arbitrary number of them because two or more unireferences can be bound to the same principal. In order to correctly capture group communication, ambient multi- and omnireferences are introduced. The following code 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;
```

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.

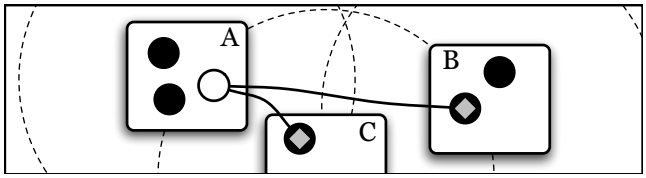

Figure 4: Ambient Omnireferences

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 dimensions discussed in this section. 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 – 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 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 from the point of view of a service designator: 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. We 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. Message sends to unireferences return futures, whose resolved result is accessible using the `when`-construct as explained in section 4.2. 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.

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. The `when`-construct from section 4.2 remains equally applicable to multifutures, only this time the closure is invoked *every time* the future is resolved with an additional return value. 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. As the number of receivers of a message sent to an omnireference is unknown, so are the number of replies. Hence, a client of the omnireference employing `when` to gather return values should not make any assumptions on the number of times the registered closure will be invoked. Omnireferences are described in more detail in the following section.

5.3 Referencing Dynamic Object Clouds

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. 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. Therefore, the principal set of such references is generally intangible. However, an application will at some point want to send a message to the cloud or grab hold of each service "currently" in the set. Below, we describe language support for fragile references to facilitate these communication and designation properties.

Communication: 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*. For example, to query the environment for all instant messengers that are in range or come in range within the next 10 seconds, one may write:

```
messengers = ambient* InstantMessenger;
when(messengers#getNickname()@10000) lambda(name) {
  println(name + " is online.");
}
```

The `@10000` annotation specifies a sustained message send of `getNickname` with a decay period of 10 seconds. This sustained message is not continually broadcast during 10 seconds. Rather, the message is multicast once to the principal set and then buffered by the omnireference for the next 10 seconds. 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". Consider the following fragile ambient omnireference hosted by a device which is part of the infrastructure of e.g. a bus station which continually "beams" a reference to an appropriate `timetableActor` into the environment for interested passengers to query from their PDA. Note how, in this scenario, the ambient reference plays the role of a service and the remote service actors play the role of interested clients.

```
(ambient* Passenger)#announce(timetableActor)@forever;
```

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 still lacks the ability to communicate with services that will bind *in the future*. Sending a sustained message with an infinite decay period to a sturdy omnireference effectively sends it to all services encountered in the past and to be encountered in the future, without sending the message twice to the same service actor.

Designation: Enumerating Object Clouds

Consider trying to enumerate the elements of the principal set of a fragile omnireference. While enumerating the elements of this set, new service actors may join the network; should these services be taken into account during the enumeration? It is equally possible that service actors which were enumerated or still have to be enumerated suddenly disjoin from the network; should those not yet enumerated be silently skipped and should those already enumerated be somehow revoked? It is clear that trying to enumerate a volatile principal set shares all of the problems associated to the iteration over a collection object that is concurrently manipulated by multiple threads.

In order to provide clean enumeration semantics, a `snapshot` operator is introduced. A `snapshot` takes any kind of ambient reference as its argument and always returns a new sturdy multireference (referred to as *the snapshot*) initialized with the set of all bound principals of its argument reference and whose cardinality equals the size of this initial principal set. A snapshot has those elasticity and cardinality behaviours which most closely resemble traditional group communication abstractions denoting a well-defined, fixed set of remote services. Because all of its principal slots are bound and sturdy, the snapshot will never bind with new services. The principal set of a fully bound sturdy multireference is by definition constant and can thus be safely enumerated by clients. It is never guaranteed that this enumeration accurately reflects the current availability of services. Elements of the snapshot can be disconnected but because the snapshot is sturdy, the bond with the service is maintained.

As an example, consider a PDA application that requires a printing service. Assume that a fragile omnireference named `printers` has been declared, denoting all available printing services on the network. In order to present the user with a list of all available services, one may enumerate a `snapshot` of the omnireference:

```
availablePrinters = snapshot(printers);
foreach printer in availablePrinters { ... }
```

5.4 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. Return values are propagated back via futures or multifutures and are accessible via the `when`-construct. 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. 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 transposed to 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. This characteristic is founded on the observation that disconnections in mobile networks are commonplace due to the volatile wireless connections.

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 still are 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 non-persistent 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 may crash but also hibernate to save power.

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 or mail address) 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 radically different messaging semantics. Omnireferences allow for direct interaction with *all* services within their scope of binding. Omnireferences are special because they allow one to denote an abstract collection of services intensionally. This collection is impossible to construct via an enumeration of individual remote references. 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 bus station example.

7. IMPLEMENTATION

As discussed in section 4, ambient references have been implemented in the actor-based ambient-oriented programming language AmbientTalk. This minimal kernel features a metaobject protocol for its actors, allowing one e.g. to change their default message receipt and message sending behaviour. It is this metaobject protocol which allows AmbientTalk to be extended from within itself with novel language features. The non-blocking futures and the `when`-construct used previously have been implemented reflectively using this MOP [12]. Ambient references themselves have been implemented reflectively on top of the more low-level event-based discovery system of the kernel explained in section 4.1. In order to comprehend the implementation of ambient references, the relevant parts of the MOP are first explained. Subsequently, it is shown how ambient references are implemented as actors whose default semantics has been altered using the MOP.

7.1 The AmbientTalk MOP

AmbientTalk actors feature first-class mailboxes; an actor has access to the messages in its own mailboxes. Moreover, an actor can monitor change in one of its mailboxes by registering mailbox observers with that mailbox. A mailbox observer is simply a closure to be invoked whenever the event it is registered for occurs. Mailboxes fire two kinds of events: one when a message is added and one when a message is deleted from the mailbox. The following code illustrates how to log all messages sent to an actor by registering a mailbox observer on the actor's inbox.

```
inbox.uponAdditionDo(
  lambda(msg) { println("received message:"+msg) })
```

An actor may intercessively add messages to and delete messages from its mailboxes. The following code snippet shows how an actor can act as a router, forwarding every message it receives to a certain `destination` actor:

```
inbox.uponAdditionDo(
  lambda(msg) {
    inbox.delete(msg);
    outbox.add(msg.setDestination(destination)) })
```

The publish-subscribe actor discovery mechanism described in section 4.1 interfaces with the MOP as follows. Actors may declare that they provide a service by invoking the method `provide(S,p)` where `S` is a service type and `p` is an optional property object. An actor may declare that it requires a service by invoking `require(S)`, where `S` is a service type. An actor that requires one or more services is notified by the kernel whenever it joins or disjoins with matching services. This notification happens via a dedicated mailbox called the *joinedbox*. Whenever a matching service joins with the actor, a *resolution* object is added to this mailbox. Conversely, whenever a joined actor disjoins, the resolution is removed from that mailbox. An actor can trap these events using mailbox observers. A resolution is a regular object with slots containing the service type which the joined actor provides, the mail address of the joined actor and the property object (if any) of the joined service actor. These mechanisms are exemplified by the following generic "sensor actor" which prints information on any service actor that becomes available. For the complete details of the metaobject protocol we refer to previous work [12].

```
require(Service);
joinedbox.uponAdditionDo(
  lambda(res) {
    mailaddr = res.provider;
    service = res.serviceType;
    println("discovered "+service+": "+mailaddr);
  })
```

7.2 Ambient Reference Mixins

Ambient references are implemented as actors. In AmbientTalk, an actor is constructed from a passive object which is known as the *behaviour* of the actor. An actor can be considered as a concurrency shell wrapped around its behaviour object. AmbientTalk objects are not instantiated from classes. Rather, objects may be created ex-nihilo or by cloning other objects. In the rest of this section, we employ the idiom of creating objects by calling methods which return new objects. The method acts as a constructor for the newly created object. In AmbientTalk, subclassing is replaced by object inheritance via delegation [19]. The combination of constructor methods and delegation allows for flexible mixin-based composition of objects, which is further illustrated below.

The behaviour of an ambient reference actor is represented by an `AmbientReference` object. This core object contains the behaviour which is common to *all* kinds of ambient references. It also defines the behaviour for an ambient reference's scope of binding.

The core object uses the MOP explained above to translate generic MOP events into more high-level events which are relevant to ambient references. The different strategies to deal with these events depend on the elasticity and the cardinality of an ambient reference and have been factored out into separate mixins. Mixins are composed via delegation. In order to create a complete ambient reference actor behaviour, first an `AmbientReference` core object is created. Next, one of three cardinality mixin objects is created whose parent is the core object. Finally, one of three elasticity mixins is created with as its parent the cardinality mixin. This elasticity mixin object together with its parent and grandparent describe the complete behaviour of one kind of ambient reference and can be used as the behaviour of an ambient reference actor. The fact that elasticity and cardinality concerns can be cleanly factored out into separate mixins strongly indicates their orthogonality.

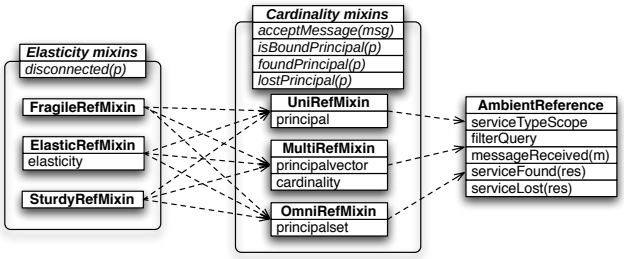

Figure 5: Ambient References as delegating mixin objects

Figure 5 illustrates the implementation of ambient references as the composition of one of three elasticity mixins, one of three cardinality mixins and an `AmbientReference` object. The figure shows objects whose names correspond to the constructor method that creates them. For example, the object named `FragileRefMixin` is created by invoking the constructor method `makeFragileRefMixin(parent)` where `parent` is a cardinality mixin object. The dotted arrows represent potential delegation. Each object has at most one parent object; the elasticity mixins are shown delegating to all cardinality mixins to emphasize that either one of them can act as their parent. This flexibility of delegation enables the modular construction of all of the ambient references introduced in section 5. The slanted, abstract methods shown above the two mixin groups are methods implemented by all of the mixins in the group. Their implementation is required by the `AmbientReference` object. In the following sections, the implementation of the core object and the two groups of mixin objects is described.

Core Ambient Reference Object

The core `AmbientReference` object which is part of any ambient reference actor's behaviour is created by calling the following constructor method:

```
AmbientReference
method makeAmbientReference(aServiceType, aQuery) {
  return new object {
    serviceTypeScope = aServiceType;
    filterQuery = aQuery;
    method init() {
      joinedbox.uponAdditionDo(this.serviceDiscovered);
      joinedbox.uponDeletionDo(this.serviceLost);
      inbox.uponAdditionDo(this.messageReceived);
      require(serviceTypeScope);
    }
    method messageReceived(msg) {
```

```
      if(!msg.isMetaMessage())
        this.acceptMessage(msg);
    }
    method serviceDiscovered(resolution) {
      if (filterQuery(resolution.properties)) {
        this.foundPrincipal(resolution.provider);
      } else {
        joinedbox.delete(resolution);
      }
    }
    method serviceLost(resolution) {
      if (this.isBoundPrincipal(resolution.provider))
        this.disconnected(resolution.provider);
    }
  }
}
```

The purpose of the above object is to translate low-level MOP events into meaningful ambient reference events. This is achieved by the three mailbox observer methods, registered with their respective mailboxes in the `init` method[2]. `init` is the first message sent to an actor after it has been created and initialized with a behaviour. After registering its mailbox observers, the ambient reference declares that it is interested in receiving notifications from the kernel pertaining to services providing the required service type (via `require(serviceTypeScope)`).

Some method calls in the bodies of the mailbox observer methods are denoted *slanted*. Such methods denote abstract methods whose implementation should be provided by an appropriate mixin object. If the above `AmbientReference` object were used stand-alone, the resulting actor would soon raise a `MessageNotUnderstood` exception when trying to invoke such an unimplemented method.

In the `messageReceived` inbox observer, the ambient reference first checks whether the incoming message is marked as a *metamessage*. Metamessages are those messages sent to the ambient reference actor itself, rather than client messages sent *via* the reference to the principal(s). An example metamessage is `snapshot` which should be handled by the ambient reference itself, rather than sent to the principal. The implementation of such meta-level operations is omitted in this description of the implementation. We only focus on how ambient references implement the forwarding of regular, base-level messages. This semantics depends on the cardinality of the ambient reference, which is why the responsibility of dealing with message acceptance is delegated to a cardinality mixin's `acceptMessage` method.

The `serviceDiscovered` joinedbox observer method is invoked whenever a matching service actor joins the network. Because the kernel discovery mechanism has no notion of filter queries, filtering services based on property objects must be done explicitly. The kernel itself only guarantees that the joined actor provides a service type which is a subtype of the requested type. Hence, the ambient reference first checks whether the properties of the new service satisfy the filter query. If so, the new service is a potential principal. Because the act of binding a new principal is again dependent on the cardinality of the ambient reference, its implementation is factored out into a cardinality mixin's `foundPrincipal` method. If the service does not satisfy the query, its resolution is discarded.

[2]In AmbientTalk, methods may be selected from an object thereby wrapping them in a closure. This makes it possible to register a method rather than an explicit closure as a mailbox observer [12].

The `serviceLost` joinedbox observer is invoked whenever a joined service actor disappears from the network. The ambient reference only has to undertake action upon such an event when the lost service is a bound principal. Whether a service is a principal or not can only be determined by the cardinality mixin of the reference. If the lost service turns out to be a bound principal, the ambient reference delegates the responsibility of dealing with this event to an elasticity mixin by invoking the abstract `disconnected` method. The following sections describe the cardinality and elasticity mixin objects, which provide implementations for the abstract methods of the ambient reference object.

Cardinality Mixins

A cardinality mixin is responsible for managing the principal(s) of an ambient reference. This includes providing and managing the principal set and handling the forwarding of received messages to the principal(s). In order to give a concrete example of the behaviour of an ambient reference, we show the complete implementation of unireferences below.

UniRefMixin

```
method makeUniRefMixin(parentObject) {
  return extend parentObject {
    principal = null;
    method acceptMessage(msg) {
      if (principal != null) {
        outbox.add(msg.setDestination(principal));
        inbox.delete(msg);
      }
    }
    method isBoundPrincipal(p) { return principal==p }
    method foundPrincipal(newPrincipal) {
      if (principal == null) {
        principal := newPrincipal;
        // forward all messages in inbox to principal
        foreach msg in inbox {
          outbox.add(msg.setDestination(principal));
          inbox.delete(msg);
        }
      }
    }
    method lostPrincipal(p) {
      principal := null;
      // return untransmitted messages to inbox
      foreach msg in outbox {
        inbox.addToFront(msg);
        outbox.delete(msg);
      }
    }
  }
}
```

A unireference's principal set consists of at most one element. Therefore, it is implemented simply as a variable named `principal` whose value determines the state of the ambient reference. If `principal` contains `null`, the reference is unbound. If it contains the mail address of a principal, the reference is bound. This variable is toggled between `null` and a principal mail address in the `foundPrincipal` and `lostPrincipal` methods.

The message forwarding behaviour of a unireference is described in the `acceptMessage` method. If the reference is bound, the incoming message is rerouted to the bound principal. If the reference is unbound, the message is kept in the inbox. The `foundPrincipal` method determines how the reference reacts upon the availability of a new candidate service provider. The appropriate behaviour depends on the state of the unireference: if it is unbound the new

service becomes the principal, if it is bound the new service is disregarded, but is kept in the joinedbox for possible later use. Upon binding to a new principal, the unireference flushes its inbox and forwards it all messages accumulated while it was unbound.

The `lostPrincipal` method can be regarded as the inverse of the `foundPrincipal` method. When the currently bound principal disconnects, the reference reverts to unbound status. It may occur that messages forwarded to the disconnected principal were not successfully transmitted. Therefore, all untransmitted messages destined for the old principal are retracted from the outbox and added to the front of the inbox again, such that they are retransmitted upon rebinding to a principal in the future. Note that the `lostPrincipal` method does not check whether the lost principal `p` equals the current `principal`. This check is already performed via the `isBoundPrincipal` invocation in the `serviceLost` method of the core ambient reference object.

Due to space limitations, we cannot completely describe the implementation of the other two cardinality mixins. An outline of their implementation is sketched below.

MultiRefMixin A `MultiRefMixin` is parameterized by its cardinality and is used to create an ambient multireference. This reference's principals are stored in an array whose size equals its cardinality. When a candidate principal is found, it is stored in the first free slot of the array. If all slots are bound, the candidate is stored for later use in the joinedbox. Upon receiving a message, the message is duplicated for each principal slot. When a principal slot is bound (i.e. does not contain `null`), the message is immediately forwarded to the bound principal. When a principal slot is unbound, the message remains in the inbox with as its destination the index of the unbound slot. Whenever a principal is bound to a slot, all messages whose destination is the slot index are removed from the inbox and sent to the new principal.

OmniRefMixin An omnireference stores its principals in a set. Upon receiving a message, the message is duplicated and sent to each principal in the set. Subsequently the reference checks whether the message is tagged as a sustained message. If not, the message is deleted from the inbox. If it is a sustained message, the ambient reference reads the decay period attached to the message and starts a timer which calls back on the omnireference after the decay period has elapsed such that the message can then be removed from the inbox. If the decay period is infinite, no timer is started and the message is left in the inbox. Whenever a principal joins the principal set, it receives a copy of all messages in the inbox.

Elasticity Mixins

An elasticity mixin specifies a policy on how to deal with disconnections of principals. Whenever a disconnection occurs, an elasticity mixin's `disconnected` method is invoked from within the core object. An elasticity mixin can decide whether the principal must actually be removed from the principal set or not. As a concrete example, the implementation of fragile references is shown below.

FragileRefMixin

```
method makeFragileRefMixin(parentObject) {
  return extend parentObject {
    method disconnected(principal) {
      this.lostPrincipal(principal);
      // try to rebind to spare candidate principals
      foreach res in joinedbox {
        if (!this.isBoundPrincipal(res.provider)) {
          this.foundPrincipal(res.provider);
          break;
        } } } }
}
```

The `disconnected` method of a fragile ambient reference immediately detaches a disconnected principal. The actual removal of the principal from the principal set is the responsibility of a cardinality mixin's `lostPrincipal` method. After having unbound the principal, the fragile reference tries to potentially rebind to spare candidate principals which are stored in the joinedbox. The `foreach` loop searches for a service provider in the joinedbox which was previously unbound and, when one is found, binds it by invoking the `foundPrincipal` method to be implemented by a cardinality mixin.

ElasticRefMixin When an elastic reference's `disconnected` method is invoked, the principal that has gone astray is marked as disconnected but is *not yet* removed from the principal set. Instead, a timer is started to call back on the reference after the elongation period. An elastic reference overrides the core object's `serviceDiscovered` method to check if a discovered service is an existing principal marked as disconnected. If so, the principal's mark is removed. When the timer calls back on the elastic reference, the latter checks whether the disconnection mark on the principal that gave rise to the notification has been removed in the mean time. If not, the elastic reference removes the principal from the principal set identical to how a fragile reference removes it.

SturdyRefMixin The implementation of a sturdy reference mixin is simple. Its `disconnected` method does nothing, thereby silently disregarding the disconnection event of the principal. Any messages sent to the principal while it is disconnected keep on being forwarded and await transmission in the outbox. The ambient reference implementation falls back on the AmbientTalk kernel's ability to properly flush untransmitted messages in the outbox when it detects that the receiver has reconnected. Because the cardinality mixins' `foundPrincipal` method checks whether a candidate principal is already bound, the principal is not inserted into the principal set multiple times when it rejoins.

Mixin Composition

Given the definition of the mixins depicted in figure 5, all different kinds of ambient references can be composed. For example, the constructor method shown below creates fragile ambient unireferences. The entries in table 1 are mere surface syntax for the invocation of such constructor functions.

```
method makeFragileUniReference(aServiceType, aQuery) {
  return new actor(
    makeFragileRefMixin(
      makeUniRefMixin(
        makeAmbientReference(aServiceType, aQuery)));
}
```

8. RELATED WORK

Related work exists in diverse research domains. First, we discuss M2MI which is a paradigm addressing the same issues as ambient references. Second, we describe the addressing and communication features of computational models and languages and their

applicability to mobile networks. Finally, we describe two middleware systems which have been explicitly designed for supporting applications deployed in mobile networks.

M2MI The design of ambient references has been inspired by the notion of a *handle* in the many-to-many invocations (M2MI) paradigm [17]. M2MI is a paradigm for building collaborative systems deployed on wireless proximal ad hoc networks. M2MI handles use Java interfaces to denote remote objects in a loosely coupled fashion. This usage of interfaces coincides with our notion of service types. M2MI distinguishes between *unihandles*, *multihandles* and *omnihandles*. Uni- and multihandles resemble sturdy ambient uni- and multireferences, although with different messaging semantics, while omnihandles resemble fragile omnireferences. An omnihandle represents all objects in communication range implementing the handle's interface. A message sent to an omnihandle means "every object out there that implements this interface, call this method" [17]. M2MI handles also employ asynchronous message passing.

Although M2MI has influenced the design of ambient references, there are some important differences. First, M2MI offers no delivery guarantees: if a message is sent to an object which is not in communication range at that time, the message is lost. Hence, message sending and delivery are not decoupled as is the case with ambient references. Only fragile omnireferences offer a similar message passing semantics, but they are augmented with sustained message delivery to widen the "temporal scope" in which messages may be delivered. In M2MI, the responsibility of guaranteed message delivery is always passed on to the application itself.

A second difference between M2MI handles and ambient references is that 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 advocates the use of callbacks to deal with return values and exceptions. As mentioned previously this solution may lead to fragmented code and requires the programmer to manually pass context information along with the method call in order to identify the source that gave rise to the callback. Ambient references employ futures (and multifutures) to deal with (multiple) return values. Furthermore, by registering closures with the future using `when`, the return value can be processed directly in the scope of invocation.

M2MI's handles are not provisional: although they may represent as yet undiscovered objects, any messages sent to this undiscovered object are lost. Moreover, as explained above, messages sent while disconnected are lost, so the communication channel defined by handles is not resilient. Omnihandles feature a transitory addressing scheme based on Java's interfaces, but are not able to retain their bond with remote objects. M2MI provides direct support for group communication via multi- and omnihandles. 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 are more sensitive to the behaviour of mobile devices and 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. The properties of mail addresses when used as "remote actor references" have already been discussed in section 4.3. To summarize, 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.

The contemporary actor language Salsa [28], which is designed for distributed computing in open networks such as the internet, introduces *universal* actors. Such actors are remotely addressable using a universal actor name (UAN). The UAN is a globally unique name for an actor, which may persist across crashes and – because actors may migrate – allows one actor to specify another actor in a device-independent fashion. Hence, Salsa's remote references based on UANs use a transitory addressing scheme. However, Salsa was not designed for infrastructure-shy mobile networks as the resolution of UANs to remote actors involves name servers. There is no discovery mechanism to get acquainted with actors that become available in the network.

E The E language [25] is designed for writing secure peer-to-peer distributed programs in open networks. The language has its roots both in the actor model and in concurrent logic/constraint programming. 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 (similar to futures and Argus' promises). E pioneered the `when` construct to deal with the resolution of promises in an entirely non-blocking, event-driven manner. The `when` construct is based on the well-known notion of *continuation* actors from the actor model [2] which are also incorporated in the ABCL family of actor languages [34].

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. It designates an object on a remote host by encapsulating the host's authentication and discovery information and a so-called *swiss number*. This is a large unguessable number which the target hosts associates with the remote object (it can be regarded as a "secret" UID).

A sturdy reference in E is, however, not a remote object reference the way a sturdy reference in AmbientTalk is. Rather, it can be used to spawn new eventual references pointing to the remote object it designates. When an object asks an E sturdy reference for a new eventual reference, it receives a promise which will either eventually resolve to the remote object, or which is resolved with an ex-

ception if reconnection to the remote host was impossible. When a host creates an E sturdy reference for one of its local objects, it may associate a lease time with the reference, denoting how long the host should maintain the mapping of the swiss number to the local object. E features a persistence model with checkpointing, so its sturdy references may persist across crashes. The salient differences between E's sturdy references and ambient references are that first, ambient references are remote object references, subsuming designation and communication in one abstraction. Second, ambient references are provisional and have the possibility to denote classes of remote services via service types. Third, ambient references automatically detect and bind to matching service actors in the proximate environment; E's sturdy references are meant to denote one unique object only, so they do not cater to transitory addressing or group communication. E's sturdy references were not designed for use in mobile networks but rather for regaining connectivity after network partitions in traditional stationary networks.

Tuple Spaces Tuple spaces as originally introduced in the coordination language Linda [14] have received renewed interest by researchers in the field of mobile computing. A tuple space is a logically shared address space between processes. Processes insert *tuples* in the tuple space which can be read by other processes on a pattern-matching basis. Because the process that adds a tuple to the tuple space does not have to address who reads the tuple, Linda is ideal for loosely-coupled distributed tasks. The notion of (asynchronous) communication without addressing is referred to as *distributed naming*: rather than using explicit addresses, programs coordinate based on patterns stored in the tuples.

Adaptations of tuple spaces for mobile computing distribute the tuple space across several devices. Linda in mobile environments (Lime) [26] is one such adaptation of Linda for mobile networks. Lime features *agents* which have their own, local *interface tuple space* (ITS). Whenever their host device makes contact with other devices in the proximity, the ITSes of different agents are transiently shared, making tuples in a remote agent's tuple space accessible while the connection lasts.

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 an agent is available to read the tuple. With respect to resilience, tuples are stored in the tuple space until an agent reads them. 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. For example, if one process places a tuple in the tuple space which denotes the "return value" of an earlier request, care has to be taken that this tuple is not accidentally read by another process that was also waiting for a return value but for a different request. Finally, the primitives which read tuples from the tuple space usually block a process until a matching tuple is available. In mobile networks, this may result in processes being blocked for extensive periods, potentially diminishing the responsiveness of the application.

ActorSpace The restrictions of mail addresses for the purposes of distributed naming have been addressed in the ActorSpace model [3, 8]. This model is a unification of concepts from both the actor model and the tuple space model of Linda. Agha and Callsen 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.

The ActorSpace model augments the actor model with *patterns*, denoting an abstract specification of a group of actors, and *actorspaces*, which are containers for actors and nested actorspaces and act as a hierarchical scoping mechanism for the resolution of patterns to mail addresses. 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 an actorspace. The `send` primitive delivers the message to a non-deterministically chosen matching actor, while the `broadcast` primitive delivers it to all matching actors. When there are no matching actors, the message send is suspended until at least one matching actor appears. This makes patterns a provisional, potentially resilient and transitory addressing mechanism.

AmbientTalk's use of service types and filter queries to delimit the scope of binding of an ambient reference is reminiscent of the ActorSpace model's notion of patterns to describe actors and actorspaces to delimit the scope of pattern resolution. 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. The model has been designed for open systems, but not for mobile networks as it introduces centralized authorities to manage the actorspaces.

Jini Sun Microsystem's Jini architecture for network-centric computing [29, 4, 30] 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 (either via a point-to-point or a broadcasting protocol) 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 provides direct support to deal with the two most apparent phenomena of mobile networks: the fact that clients and service providers may join with and disjoin from the network at any time, without any prior warning. As already mentioned, spontaneous discovery is taken care of via a periodic broadcasting protocol man-

aged by the lookup services. To deal with unheralded disconnections, Jini employs a lease-based referencing mechanism: services must explicitly renew their lease with the lookup service; if they cannot, the lookup service will remove the service advertisement such that it doesn't provide stale information. Likewise, clients should interact with services on the basis of a lease such that a service may reclaim any resources allocated for the client session whenever either one disjoins from the network.

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. It may also shield a client from the communication protocol used and even from service upgrades [27]. Hence, Jini's architecture is flexible enough to cater for 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.

STEAM Scalable Timed Events and Mobility (STEAM) is an event-based middleware designed for supporting collaborative applications in mobile ad hoc networks [24]. Like AmbientTalk, STEAM shuns the use of centralized components such as lookup and naming services to void any dependencies of mobile devices on a common infrastructure. STEAM builds upon the observation that the physically closer an event consumer is located to an event producer, the more interested it may be in those events. It allows events disseminated by producers to be filtered based on e.g. geographical location using *proximity filters*. Examples include traffic management scenarios where cars notify one another of nearby accidents and traffic lights automatically signal their status to cars near a road intersection. Proximity filters operate on *proximities*, which may be absolute or relative (i.e. a relative proximity denotes a surrounding area relative to a mobile node). STEAM offers a location service which uses sensor data and GPS coordinates to determine the geographical location of nodes in the network.

STEAM proximities relate to ambient omnireferences in the sense that both may be used for event dissemination to a select set of services within proximity. However, STEAM is publish-subscribe middleware and has no notion of remote object references. On the other hand, STEAM's proximity mechanism to denote "scope of binding" is more elaborate than that of omnireferences. AmbientTalk does not include a location service but rather uses UDP multicasts over WiFi to discriminate proximate from distant services. We regard the incorporation of the notion of STEAM proximities into ambient references via service types representing locations as interesting and essential future work.

9. LIMITATIONS AND FUTURE WORK

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

Service Discovery

AmbientTalk's discovery mechanism based on service types is very similar to e.g. Jini's discovery mechanism which is based on interface types. A common critique on such discovery mechanisms is that they lack rich service representations, the ability to specify constraints on the search and that they do not support inexact matches. Alternative discovery approaches have been proposed based on agents and ontology reasoning engines [11]. We are aware of the limited expressive power of the discovery mechanism of ambient references. 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 our work on 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. For an overview of the state of the art in service discovery protocols, we refer to the survey paper by McGrath [23].

Security

In mobile ad hoc networks, security issues are exacerbated because the network is usually not administered, nor is there any central authority e.g. to validate the identity of hosts and services. From the perspective of ambient references, two prime security issues become apparent. First, how can an ambient reference guard itself from binding with (and sending information to) a compromised or malicious service? Second, how can a service protect itself from being bound to malicious ambient references? Both problems require the proper authorization of clients and services. Asymmetric public/private key encryption algorithms may in part alleviate these problems, using public-key encryption to ensure confidentiality and public-key digital signatures for authentication. These mechanisms are definitely not a panacea for all security issues, and sometimes require centralised authorities (e.g. public key infrastructures) which are highly undesirable in ad hoc mobile networks.

Mechanisms to deal with trust and authorization could be incorporated into the ambient reference design space via the scope of binding. For example, if one can discriminate between trusted and distrusted services, the scope of binding can be used to disallow a reference from binding to distrusted ones.

Customized Ambient References

One important observation with respect to the mixin-based implementation of ambient references sketched in section 7 is that the abstract methods referred to by the core ambient reference object define a *protocol*, which, when made available to the programmer is the basis for what one might call a *meta-ambient reference protocol*. If an AmbientTalk programmer can write his own modular mixin objects implementing the required abstract methods, this allows him to specify entirely new, application-specific addressing and communication abstractions, within the framework provided by ambient references. Actually, this is already perfectly possible as ambient references have been implemented reflectively and are as such plain accessible AmbientTalk objects. However, it remains an open question as to what level of detail the ambient references implementation should be opened up to the programmer and what kinds of adaptations lead to useful application-specific ambient references.

Distributed Garbage Collection

For remote references to remain valid, the host of the designated remote object is responsible for keeping an export table mapping UIDs to local object references. As a consequence, exported objects cannot simply be reclaimed, giving rise to the notion of distributed garbage collection. DGC requires a set of hosts to cooperatively clean up 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 are 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.

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.

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 transitory addressing scheme which is device-independent and persistent. 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.

11. REFERENCES

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

[2] AGHA, G. Concurrent object-oriented programming. *Communications of the ACM 33*, 9 (1990), 125–141.

[3] AGHA, G., AND CALLSEN, C. J. Actorspace: An open distributed programming paradigm. In *Proceedings of the 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices* (1993), pp. 23–32.

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

[5] 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.

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

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

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

[9] 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.

[10] 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.

[11] CHEN, H., JOSHI, A., AND FININ, T. Dynamic service discovery for mobile computing: Intelligent agents meet jini in the aether. *Cluster Computing 4*, 4 (Oct 2001), 343–354.

[12] 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. To Appear.

[13] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC language: a holistic approach to networked embedded systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2003).

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

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

[16] 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.

[17] 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.

[18] LEVIS, P., MADDEN, S., GAY, D., POLASTRE, J., SZEWCZYK, R., WOO, A., BREWER, E. A., AND CULLER, D. E. The emergence of networking abstractions and techniques in TinyOS. In *Proceedings of the first Symposium on Networked Systems Design and Implementation (NSDI 2004)* (March 29-31 2004), USENIX, pp. 1–14.

[19] LIEBERMAN, H. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented Programming Systems, Languages and Applications* (1986), ACM Press, pp. 214–223.

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

[21] 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.

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

[23] 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.

[24] MEIER, R., AND CAHILL, V. Exploiting proximity in event-based middleware for collaborative mobile applications. In *Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'03)* (2003).

[25] 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.

[26] 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.

[27] SEN, R., HANDOREAN, R., HACKMANN, G., AND ROMAN, G.-C. An architecture supporting run-time upgrade of proxy-based services in ad hoc networks. In *Proceedings of the International Conference on Pervasive Computing and Communications, PCC'04* (June 21-24 2004), pp. 689–695.

[28] VARELA, C., AND AGHA, G. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not. 36*, 12 (2001), 20–34.

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

[30] WALDO, J. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)* (2001), p. 9.

[31] 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.

[32] WILKES, C., AND LEBLANC, R. Rationale for the design of aeolus: A systems programming language for an action/object system. In *Proceedings of the IEEE CS 1986 International Conference on Computer Languages* (New York, Oct. 1986), IEEE, pp. 107–122.

[33] YONEZAWA, A., Ed. *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press, 1990.

[34] 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.

APPENDIX

A. AMBIENTTALK EXAMPLE CODE

The AmbientTalk equivalent of all code examples and mixin implementations of the paper are shown below. The syntax is discussed in more detail in previous work [12].

Section 4.1

```
servicetype(InstantMessenger, Service);

makeInstantMessenger(nickname) :: {
  actor(object({
    init() :: { provide(InstantMessenger) };
    ...
  }))
}
```

Section 4.2

```
nameFuture : anInstantMessenger#getNickname();

when(nameFuture, lambda(name) -> {
  display(name, " is online.")
})
```

Section 5

```
anInstantMessenger : ambientFragileUni(InstantMessenger);
```

Section 5.1

```
makeInstantMessenger(id) :: {
  actor(object({
    init() :: { provide(InstantMessenger,object({accountid :: id})) };
    ...
  }))
}

ambientFragileUni(InstantMessenger, lambda(m)->{ m.accountid.equals(buddyId) });
```

```
group : vector.newWithSize(10);
for(i:1, i<=10, i:=i+1, {
  group.add(ambientFragileUni(InstantMessenger));
})

allMessengers : ambientFragileOmni(InstantMessenger);
tenMessengers : ambientFragileMulti(10, InstantMessenger);
```

Section 5.3

```
messengers : ambientFragileOmni(InstantMessenger);
when(sustain(messengers#getNickname,[],10000), lambda(name)->{
  display(name, " is online.")
})

sustain(ambientFragileOmni(Passenger)#announce,[timetableActor],forever);

availablePrinters : printers#μsnapshot();
when(availablePrinters#μenumerate(), lambda(collection) -> {
  collection.iterate(lambda(printer) -> { ... })
})
```

Section 7.1

```
inbox.uponAdditionDo(
  lambda(msg)->{ display("received message:",msg) })

inbox.uponAdditionDo(
  lambda(msg)->{
    inbox.delete(msg);
    outbox.add(msg.setDestination(destination)) })

require(Service);
joinedbox.uponAdditionDo(
  lambda(res)->{
    mailaddr : res.provider;
    service : res.serviceType;
    display("discovered ",service,": ",mailaddr);
  })
```

AmbientReference

```
makeAmbientReference(aServiceType, aQuery) :: {
  object({
    serviceTypeScope : aServiceType;
    filterQuery : aQuery;
    init() :: {
      joinedbox.uponAdditionDo(this.serviceDiscovered);
      joinedbox.uponDeletionDo(this.serviceLost);
      inbox.uponAdditionDo(this.messageReceived);
      require(serviceTypeScope)
    };
    messageReceived(msg) :: {
      if(!msg.isMetaMessage(),
        this.acceptMessage(msg))
    };
    serviceDiscovered(resolution) :: {
      if(filterQuery(resolution.properties), {
        this.foundPrincipal(resolution.provider)
      }, {
        joinedbox.delete(resolution)
      })
    };
    serviceLost(resolution) :: {
      if(this.isBoundPrincipal(resolution.provider),
        this.disconnected(resolution.provider))
    }
  })
}
```

UniRefMixin

```
makeUniRefMixin(parentObject) :: {
  extend(parentObject, {
    principal : void;
    acceptMessage(msg) :: {
      if(!is_void(principal), {
        outbox.add(msg.setDestination(principal));
        inbox.delete(msg)
      })
    };
    isBoundPrincipal(p) :: { principal == p };
    foundPrincipal(newPrincipal) :: {
      if(is_void(principal), {
        principal := newPrincipal;
        // forward all messages in inbox to principal
        inbox.iterate(lambda(msg)->{
          outbox.add(msg.setDestination(principal));
          inbox.delete(msg)
        })
      })
    };
    lostPrincipal(p) :: {
      principal := void;
      // return untransmitted messages to inbox
      outbox.iterate(lambda(msg)->{
        inbox.addToFront(msg);
        outbox.delete(msg);
      }) } })
}
```

FragileRefMixin

```
makeFragileRefMixin(parentObject) :: {
  extend(parentObject, {
    disconnected(principal) :: {
      this.lostPrincipal(principal);
      // try to rebind to spare candidate principals
      found : joinedbox.findFirst(lambda(res)->{
        !this.isBoundPrincipal(res.provider)
      });
      if(!is_void(found),
        this.foundPrincipal(found.provider))
    } })
}
```

Section 7.2

```
makeFragileUniReference(aServiceType, aQuery) :: {
  actor(
    makeFragileRefMixin(
      makeUniRefMixin(
        makeAmbientReference(aServiceType, aQuery))))
}
```

Ambient Reference Constructor Example

```
// variable argument function: if one parameter is given, default filter is used
ambientFragileUni@args :: {
  serviceType : args[1];
  filterQuery : if(size(args) < 2, lambda(props)->{ true }, args[2]);
  makeFragileUniReference(serviceType, filterQuery)
}
```