Ambient-Oriented Programming

Jessie Dedecker* Tom Van Cutsem*
Stijn Mostinckx† Theo D'Hondt Wolfgang De Meuter
Programming Technology Laboratory
Department of Computer Science
Vrije Universiteit Brussel, Belgium
jededeck | tvcutsem | smostinc | tjdhondt | wdmeuter@vub.ac.be

ABSTRACT

A new field in distributed computing, called Ambient Intelligence, has emerged as a consequence of the increasing availability of wireless devices and the mobile networks they induce. Developing software for such mobile networks is extremely hard in conventional programming languages because the network is dynamically defined. This hardware phenomenon leads us to postulate a suite of characteristics of future *Ambient-Oriented Programming* languages. A simple reflective programming language kernel, called AmbientTalk, that meets these characteristics is subsequently presented. The power of the reflective kernel is illustrated by using it to conceive a collection of high level tentative ambient-oriented programming language features.

Categories and Subject Descriptors

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

General Terms

Design, Languages

Keywords

ambient intelligence, mobile networks, actors, language kernel

1. INTRODUCTION

Software development for mobile devices is given a new impetus with the advent of *mobile networks*. Mobile networks surround a mobile device equipped with wireless technology and are demarcated dynamically as users move about. Mobile networks turn the applications running on mobile devices from mere isolated programs into smart applications that can cooperate with their environment. As such, mobile networks take us one step closer to the world of ubiquitous computing envisioned by Weiser [44]; a world where (wireless) technology is gracefully integrated into the everyday lives of its users. Recently, this vision has been termed *Ambient Intelligence* (AmI for short) by the European Council's IST Advisory Group [21].

Mobile networks that surround a device have several properties that distinguish them from other types of networks. The most important ones are that connections are volatile (because the communication range of the wireless technology is limited) and that the network is open (because devices can appear and disappear unheraldedly). This puts extra burden on software developers. Although low-level system software and networking libraries providing uniform interfaces to these wireless technologies (such as JXTA [17] and M2MI [26]) have matured in the last couple of years, developing application software for mobile networks still remains difficult. One of the main reasons for this is that current-day programming languages lack abstractions to deal with the mobile hardware characteristics. For instance, in traditional programming languages failing remote communication is usually dealt with using a classical exception handling mechanism. This results in application code polluted with exception handling code because mobile network failures are the rule rather than the exception. Observations like this justify the need for a new *Ambient-Oriented Programming* paradigm (AmOP for short) that consists of programming languages that explicitly incorporate potential network failures in the very heart of their basic computational steps.

As very little experience exists in writing applications that fully explore the potential of mobile networks, it is hard to come up with a definition of AmOP based on software engineering requirements. Therefore, our research has focussed on the hardware phenomena that distinguish mobile networks from existing stationary networks. These phenomena are listed in section 2.1 and form the basis from which we distill a number of fundamental programming language characteristics that define the AmOP paradigm. These characteristics are the topic of section 3. A concrete scion of the AmOP paradigm — called AmbientTalk — is presented starting from section 4. AmbientTalk was conceived as an improvement on the classical actor model that is particularly focused on mobile networks. Moreover, AmbientTalk was designed as a minimalist extensible language kernel be-

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

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

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

cause of two reasons. First, it was our goal to investigate the impact of the new hardware context on existing distributed programming language features. Second, the lack of extensive software engineering experience for AmI fosters an experimental approach to distill the list of language features necessary for AmI. Both goals are achieved by conceiving AmbientTalk as a reflectively extensible kernel that allows us to explore the boundaries of the paradigm. A description of the reflective features is presented in section 4.3. Finally, section 5 presents a concrete experiment we conducted in a particular extension of AmbientTalk. The experiment comprises the implementation of an ambient peer-to-peer instant messaging application deployed on smart phones.

2. MOTIVATION

In the context of mobile networks, one sometimes makes a distinction between nomadic and ad hoc distributed systems depending on whether a shared infrastructure is used to support the mobile communication [30]. Nomadic systems use such an infrastructure (e.g. cellular phones hopping from one cell to another) whereas ad hoc ones do not (e.g. two PDA's that encounter each other). The technical hardware properties of the devices which constitute ad hoc and nomadic distributed systems engender a number of phenomena that have to be dealt with by the middleware and/or distributed programming language processors. We summarize these hardware phenomena below and describe how existing programming languages and middleware fail to deal with them. This forms the main motivation for our work.

2.1 Hardware Phenomena

With the current state of commercial technology, mobile devices are often characterised by having scarcer resources (such as lower CPU speed, smaller memory and limited battery) than traditional hardware. However, we cannot help but notice that in the last couple of years, mobile devices and full-fledged computers like laptops are blending more and more. That is why we do not consider these restrictions as fundamental to the AmOP paradigm as we consider the following phenomena to be:

- **Connection Volatility.** Two processes that perform a meaningful task together on two cooperating devices cannot assume a stable connection. The limited communication range of the wireless technology combined with the fact that users can move out of range can result in broken connections. However, upon re-establishing a broken connection, users typically expect the task to resume. In other words, they expect the task to be performed in the presence of a volatile connection.

- **Ambient Resources.** If a user moves with his mobile device, remote resources become dynamically (un)available in the environment because the availability of a resource may depend on the location of the device. This is in contrast with stationary networks in which references to remote resources are obtained based on the explicit knowledge of the availability of the resource. In the context of mobile networks, the resources are said to be ambient.

- **Autonomy.** Most distributed applications today are developed using the client-server approach. The server often plays the role of a "higher authority" which coordinates interactions between the clients. In mobile networks, and especially in mobile ad hoc networks, a connection to such a "higher authority" is not always possible. Every device acts as an autonomous computing unit.

- **Natural Concurrency.** In theory, distribution and concurrency are two different phenomena. For instance in a client-server setup, a client might wait for the results of a request to the server in order to resume its computation. Hence, in theory a distributed system is not necessarily a concurrent one. However, even in the extreme case where both communicating devices run a single threaded program, their autonomy implies that the resulting task is a concurrent one. Moreover, the trend of software getting ever more multi-threaded will also manifest itself on mobile devices. As a result, concurrency is a natural phenomenon in software running on mobile networks.

2.2 Distributed Languages

To the best of our knowledge no distributed language has been designed to specifically deal with the characteristics of mobile hardware just described. Existing distributed languages can be categorised as languages designed for local area networks and languages that have been designed for open networks, such as the internet.

2.2.1 Languages for Local Area Networks

A number of distributed languages have been proposed, which target local area networks. Some of them are based on synchronous (hence blocking) communication primitives, such as Emerald [23] and Obliq [7], while others, like ABCL/f [36] and Argus [27, 28] promote an intermediate form based on futures [19]. These communication mechanisms are feasible for reliable networks, where failures are the exception, but harm the autonomy when used in high latency networks, such as mobile networks.

2.2.2 Languages for Open Networks

Some distributed languages, such as Janus [25], Salsa [42] and E [31], are based on the actor model. The actor model is based on pure asynchronous communication, which preserves the autonomy of devices in the context of high latency and failures. However, these languages offer no support to discover ambient resources or to deal with consistency among autonomous computing units.

2.3 Distributed Middleware

An alternative to distributed languages is middleware. Over the past few years a lot of research has been invested in middleware for nomadic and ad hoc distributed systems [30]. This bulk of research can be categorized into several groups.

2.3.1 RPC-Based Middleware

Alice [18] and DOLMEN [33] are attempts to make CORBA feasible for supporting nomadic distributed systems. These attempts focussed mainly on making heavyweight ORBs suitable for the lightweight devices and on improving the resilience of the IIOP protocol to failing communication.

Other approaches adapt the RPC protocol by supporting queuing of RPCs [22] or enabling rebinding of resources [35]. These approaches work well when connections are lost for a short time, but do not address disconnections over longer periods of time.

2.3.2 Publish-Subscribe Middleware

Another, more recent, branch of middleware for mobile computing is based on the adaptation of the publish-subscribe paradigm [14] to cope with the characteristics of mobile computing [8, 6, 9]. Such middleware allows asynchronous communication, but has the disadvantage of requiring callbacks to handle results, which clutters the code and makes the program less understandable.

2.3.3 Tuple Space Based Middleware

In the past few years middleware has been proposed [32, 10, 29, 15] for mobile computing based on tuple spaces [16]. A tuple space acts as an intermediate data structure in which processes can publish and query tuples to communicate asynchronously with one another. Most research on tuple spaces for mobile computing consists of distributing the tuple space over a set of devices. Although tuple spaces are an interesting communication paradigm for mobile computing, the paradigm does not integrate well with the object-oriented paradigm because communication is achieved by placing data in a tuple-space as opposed to sending messages to objects.

2.3.4 Data Sharing-Oriented Middleware

Another branch of middleware tries to maximize the autonomy of mobile devices. In most approaches, such as Coda [34], Bayou, [37], Rover [22] and XMiddle [46], this is achieved by introducing weak replica management facilities in the middleware. Due to the connection volatility, replicas are not always synchronized. This can lead to a series of conflicts, which are application-specific and must be resolved at the application level.

2.4 Summary

The current state of the art in middleware does not address all the important characteristics that are encountered when developing a nomadic or ad hoc distributed system. Regarding distributed languages there have been a number of proposals suitable for open distributed networks, but they have not yet been applied in the context of mobile computing and do not deal with all the hardware phenomena described in section 2.1.

3. AMBIENT-ORIENTED PROGRAMMING

In the same way that referential transparency can be regarded as a defining property for pure functional programming, this section presents a collection of language design characteristics that define the boundaries of the ambient-oriented programming paradigm. These characteristics are directly derived from the hardware phenomena we summarized in section 2.1. Until now, it seems that the object-oriented paradigm is the most successful one w.r.t. dealing with distribution and its induced concurrency because it successfully aligns encapsulated objects with concurrently running distributed software entities [4]. Therefore, our most basic research assumption is that ambient-oriented programming languages necessarily are concurrent distributed object-oriented programming languages. However, ambient-oriented programming languages differ from conventional distributed concurrent object-oriented programming languages in at least one of the following four ways:

3.1 Prototype-based Object Models

As a consequence of parameter passing in the context of remote messages, objects are copied back and forth between remote hosts. Since an object in a class-based programming language cannot exist without its class, this copying of objects implies that classes have to be copied as well. However, a class is – by definition – an entity that is conceptually shared by all its instances. From a conceptual point of view there is only one single version of the class on the network, containing the shared class variables and method implementations. Hence, copying classes over the network causes state consistency problems because objects residing on different machines can independently update a class variable of "their" copy of the class. Moreover, a device might upgrade to a new version of a class thereby "updating" its methods. These are both classical distributed state consistency problems and solving them requires replication machinery. However, in our hardware context consisting of autonomous devices that are connected in a volatile fashion, solving this problem poses some fundamental paradigmatic problems.

By definition, classes impose a sharing relation upon all their instances. This relation is established at object creation time and remains *implicit* throughout the lifetime of *all* its instances. However, because of independent class updates performed by autonomous disconnected devices, two instances of the same class can unexpectedly exhibit different behaviour. In other words, the implicit relation becomes explicitly detectable. Existing class-based languages do not offer programmers the means to deal with this phenomenon since classes are usually not *fully* reified in the language. For instance, the instance-of link between classes and objects is usually not made explicit in the language precluding transmitted objects from changing their class to a more suitable version upon arrival. Worse, upon dealing with inconsistent versions of the same class, no application-independent rule exists to prefer one class over the other. To allow programmers to specify how such conflicts are to be resolved, the only viable solution is to *fully* reify classes and the instance-of relation. However, this is easier said than done. Even in the absence of wireless distribution, languages like Smalltalk and CLOS — which do not fully reify their class system — already illustrate that a serious reification of classes and their relation to objects results in extremely complex meta machinery.

A much simpler solution consists of getting rid of classes and the sharing relation they impose on objects altogether. This is the paradigm defined by prototype-based languages like Self [40]. In these languages objects are *conceptually* entirely idiosyncratic such that the above problems do not arise. Sharing relations between different prototypes can still be established (such as e.g. traits [39]) but the point is that these have to be explicitly encoded by the programmer at all times. Surely, a runtime environment can optimise things by sharing properties between different objects. But such a sharing is not part of the language definition and can never be detected by objects. For these reasons, we have decided to select prototype-based object models for

ambient-oriented programming. Notice that this confirms the design of existing distributed programming languages such as Emerald, Obliq, dSelf and E which are all classless.

3.2 Non-Blocking Communication Primitives

The fact that every hardware device is an autonomous computational entity (inducing natural concurrency) combined with the fact that connections are volatile, implies the necessity for non-blocking communication primitives. Blocking communication is a source of (distributed) deadlocks [43]. Deadlocks and distributed deadlocks in local networks are not considered to be that harmful, since the cause of the deadlock can be easily debugged with contemporary remote debugging environments. However, in mobile networks, not all parties are necessarily available for communication making the resolution of deadlocks extremely hard. Another, more important consideration when designing a concurrency model for a language that is to run on mobile networks, is that the communication mechanism should minimize the duration resources are locked. This is very important, because the extremely high latency of communication (over volatile connections) in mobile networks would diminish the availability of resources. Indeed, having blocking communication primitives would imply a program or device to block upon encountering unstable connections or temporary unavailability of another device. This has previously been remarked on several occasions [30, 9, 32]. We thus conclude that an ambient-oriented concurrency model is a concurrency model without blocking communication primitives.

Quite often, the issue of non-blocking communication is confused with asynchronous message sending. Asynchronous message sending implies that the send operation is non-blocking, but tells us nothing about the (possibly implicit) receive operation. A typical example of asynchronous send operations combined with blocking receive operations is found in the tuple-space based middleware (discussed in section 2.3.3), which provide explicit, blocking receive operations on the tuple-space.

3.3 Reified Communication Traces

Non-blocking communication (both send and receive) combined with the autonomy of the communicating devices implies that they will have to foresee *some* form of handshaking given the fact that these devices are performing a meaningful task together. Since the communication is non-blocking, both senders and receivers will continue their execution irrespective of what happened after a message send. This means that the parties might end up in a state that is not consistent with the semantics of whatever the task it is that they are solving. Whenever such an inconsistency is detected, the parties must be able to restore their state to whatever previous consistent state they were in, such that they can decide what to do based on the final consistent state they agreed upon. Examples of the latter could be overruling one of the two computations or deciding together on a new state with which both parties can resume their computation. Therefore, a programming language in the ambient-oriented paradigm will have to provide us with reversibility provisions giving programmers a way to manipulate their execution state based on an *explicit representation* (i.e. a reification) of the communication details that led to the inconsistent state. The explicit representation will allow them to take the appropriate actions to reverse (part of) the computation. Notice that any implicit way to prevent the communicating parties from ending up in an inconsistent state implies that communication primitives are blocking, which was precluded above. Having an explicit reified representation of whatever communication that happened, allows a device to properly recover from an inconsistency by reversing part of its computation.

Several degrees of message delivery guarantee can be associated with non-blocking communication. For example, in the many-to-many invocations library [26], where all communication occurs via asynchronous messages, there are no delivery guarantees. When a message is sent and there is no process listening for messages, the message is lost. Such communication paradigm is lightweight with respect to the usage of resources and is suitable when no delivery guarantees are to be met. On the other end of the spectrum there is the actor model, where all asynchronous messages that are sent must eventually be received [1]. Such an approach is perhaps feasible when there are abundant resources, but in the context of mobile computing, where devices have scarce resources, it is clear that such an approach is not practicable. This shows that there is no single "right" message delivery guarantee policy because a tradeoff will have to be made based on the requirements of the application and on the available resources. Programming languages belonging to the ambient oriented paradigm should make this tradeoff possible instead of imposing a single strategy. Explicit control over the communication traces allows one to make the tradeoff between different delivery guarantees.

3.4 Ambient Acquaintance Management

The fact that hardware devices are autonomous, combined with the fact that resources are dynamically detected as the devices are roaming means that all devices potentially have the same capabilities to interact with each other directly without relying on a third party. This is in contrast to client-server communication models where clients usually interact through the mediation of a server (such as is the case with chat servers or white boards). The fact that communicating parties do not need an explicit reference to each other (whether directly or indirectly through a server) requires what is known as distributed naming [16]. For example, in tuple-space based middleware this property is achieved, because a process can publish data in a tuple space, which can then be consulted by the other processes based on a pattern matching basis. Another example is many-to-many invocations [26], where broadcasts to all objects implementing a certain interface can be expressed. Distributed naming is especially important in the context of ad hoc distributed systems, because it provides a mechanism to communicate without knowing the address of an ambient resource.

We are not arguing that all ambient-oriented applications must be based on distributed naming. It is perfectly possible that a programmer (or even a suite of running processes) sets up a server for the purposes of a certain application. However, an ambient-oriented programming language should allow applications to rely on distributed naming if it is required. In other words, the acquaintances of an object must be dynamically manageable.

3.5 Discussion

Having analysed the implications of the hardware phe-

nomena on the design of programming languages, we have come up with the above four characteristics. We will henceforth refer to programming languages that adhere to them as *Ambient-oriented Programming Languages*. Surely, it is impossible to prove that these are strictly necessary characteristics for writing the applications we target. After all, AmOP does not transcend Turing equivalence. However, we do claim that an AmOP language will greatly enhance the construction of such applications because their distribution characteristics are designed with respect to the hardware phenomena presented in section 2.1. AmOP languages incorporate temporal disconnections and evolving acquaintance relationships in the heart of their computational model.

The current state of the art in distributed languages does not conform to all characteristics of AmOP. On the one hand, languages for local area networks do not have the non-blocking communication characteristic. On the other hand, languages for open networks usually have the non-blocking communication characteristic, but do not allow for ambient acquaintance management and are not equipped with reversibility provisions.

4. THE AMBIENTTALK KERNEL

Now that we have established the characteristics of the AmOP paradigm, the stage is set for the introduction of an exemplar ambient-oriented programming language, called AmbientTalk. AmbientTalk was designed as a minimal reflectively extensible language kernel. We begin by explaining the essential characteristics of its object model.

4.1 The Object Model

AmbientTalk has a modern object model that was built on previous research especially regarding its concurrency model which was heavily based on ABCL/1 [45]. This model is basically a marriage between two extremes in object concurrency, to wit a functional model based on actors and messages, and an imperative model based on threads that run through multiple objects. The marriage proposed in ABCL/1 features active objects which consist of a perpetually running thread, updateable state and a message queue. These concurrently running active objects communicate with each other by asynchronous message passing. Upon reception, messages are scheduled in the object's message queue and are processed by the object's thread one by one. By excluding simultaneous message processing, race conditions on the updateable state are avoided. The merit of this model lies in the fact that it unifies imperative object-oriented and concurrent programming without suffering from omnipresent race conditions.

In order to avoid the fact that every single object has to be equipped with rather heavyweight concurrency provisions, and to preclude that every single message has to be thought of as a concurrent one, a more fine-grained object model that distinguishes between active and passive (i.e. ordinary) objects is desired. This allows programmers to deal with concurrency only when strictly necessary (i.e. when considering semantically concurrent and/or distributed tasks). Since passive objects are not equipped with an execution thread, the "currently running" thread simply runs from the sender into the receiver, thereby implementing synchronous message passing. However, when combining active objects with passive ones, it is important to ensure that a passive object is never shared by two different active ones because this easily leads to race conditions. AmbientTalk's object model avoids this by obeying the following principles:

- *Containment* A passive object is never shared by two active ones. Every passive object is therefore owned by exactly one active object. The only thread that can enter the passive object is the thread of the active object in which the passive one is contained.
- *Parameter Passing* When an asynchronous message is sent to an active object (either from within an active or a passive object), passive objects may be exchanged as arguments and return values. In order not to violate the containment principle, a passive object that crosses the boundary of its active container is therefore always passed by copy. Active objects are simply passed by reference.

This pragmatic marriage between the functional actor model, the imperative thread model and the ordinary passive prototype-based model was chosen as the basis for AmbientTalk's distribution model. The fact that messages sent to passive objects are always synchronous messages is not reconcilable with the non-blocking communication characteristic derived in section 3. Therefore, active objects are the unit of distribution in AmbientTalk. Hence, applications in AmbientTalk are conceived as suites of active objects deployed on different devices. Each active object can contain a plethora of passive ones but these can never be exposed to the network due to the rules specified above.

4.2 First-class Mailboxes

AmbientTalk's concurrent object model presented above was explicitly designed to meet two of the characteristics presented in section 3 (namely a prototype-based object model supporting non-blocking communication). However, with respect to the other two characteristics, the model presented so far still has some limitations which it directly inherits from the original actor model as proposed by Hewitt and Agha [20, 2]:

- **Ambient Acquaintance Management**: The model does not support the *Ambient Acquaintance Management* characteristic of the AmOP paradigm, because actors can only gain acquaintances through other actors. The ActorSpace model [5], an extension of the actor model, enables distributed naming by introducing an actor grouping mechanism, named *spaces*. However, these spaces are managed by centralized authorities, which is infeasible in a mobile computing setting.
- **Reified Communication Traces**: Actors and ActorSpaces do not support reified communication traces. As a consequence, it is e.g. impossible to introduce language constructs which retract messages that were sent, but not yet transmitted. This kind of flexibility is sometimes needed to resolve conflicts in the case of network partitions as argued in section 3.3.

To enable these two properties, AmbientTalk replaces the single message queue of the original actor model by a system of first-class mailboxes which is described below. AmbientTalk's first-class mailboxes are based on a formal extension of the actor model, called the ambient actor model [12]. In the remainder of the paper, the term *actor* will be used to refer to an active object (as described above) that is equiped with first-class mailboxes.

4.2.1 Reified Communication Traces

When scrutinising the communication structure of an actor, we observe four types of messages. Each type is stored in a dedicated mailbox by AmbientTalk. The `in` mailbox stores incoming messages, which have been received without having been processed yet. Those that have been processed are transferred from the `in` mailbox to the `rcv` mailbox. The `out` mailbox stores those messages which an actor has sent but which are not yet delivered. Upon successful delivery, a message is transferred from the `out` mailbox to the `sent` mailbox. Notice that the combined behaviour of the `in` and `out` mailboxes was already implicitly present in the original actor model in the form of a simple message queue. It is the basis for non-blocking communication.

Apart from the four predefined mailboxes of an actor, every actor can create its own custom mailboxes. Operators exist to explicitly add and delete messages that reside in both the predefined and the custom mailboxes. Moreover, the changes in a mailbox can be monitored by registering observers. These mechanisms provide us with all facilities necessary to fully reify an actor's communication traces. For example, by removing a message from the `out` mailbox it can be stopped from being delivered. In the same vein, a message can be removed from the `in` mailbox to prevent it from being processed by the actor.

Conceptually, the mailboxes `rcv` and `sent` allow one to have a peek in the past of the communication history of an actor. Likewise, the mailboxes `in` and `out` represent its continuation, because they contain the messages the actor will process and deliver in the future. These four explicit mailboxes provide a gate to the past and the future of the actor's communication state; i.e. the reified communication traces that have been prescribed by the AmOP paradigm.

4.2.2 Ambient Acquaintance Management

In section 3.4 we argued that an ambient acquaintance management facility forms an essential ingredient of an AmOP language. To achieve this, AmbientTalk actors have four additional predefined mailboxes named `joined`, `disjoined`, `required` and `provided`.

An actor that wants to make itself available for collaboration can advertise itself by placing one or more descriptive tags (e.g. strings) in its `provided` mailbox. Conversely, an actor that needs other actors in its ambient places such descriptive tags in its `required` mailbox. When two or more actors enter one another's communication range and have a corresponding descriptive tag in their mailboxes, the mailbox `joined` of the actor that required the collaboration is updated with a *resolution*. Such a resolution is a pair consisting of the matching tag and a reference to the actor that provided the tag. Conversely, when two actors with a corresponding tag in their mailboxes are pulled out of communication range, the resolution is moved from the `joined` mailbox to the `disjoined` mailbox. This mechanism allows an actor not only to detect new acquaintances in its ambient, but also to detect when they have disappeared from the ambient. It is AmbientTalk's technical realisation of the ambient acquaintance management characteristic discussed in section 3.4.

4.3 AmbientTalk as a Reflective Kernel

As explained before, AmbientTalk is a minimalist reflective kernel that has the essential built-in features to deal with the AmOP characteristics outlined in section 3. The kernel basically consists of the above object model together with the system of eight built-in mailboxes. An actor's mailboxes are "causally connected" to the outer world as their content reflects the history and future of its computational state and the hardware constellation that currently surrounds it.

A mailbox is reified as a passive object associated with exactly one actor and can (like any other passive object) only be passed by copy to other actors[1]. Although a mailbox is private to a single actor, it is manipulated by two different entities. On the one hand, it can be explicitly manipulated by its owner due to AmbientTalk's reflective properties. On the other hand, the underlying actor system also updates the mailbox when certain low-level events occur, for example when a message is transmitted from the `out` mailbox of one actor to the `in` mailbox of another. The implementation of AmbientTalk ensures that the manipulation of mailboxes by these two entities cannot occur concurrently in order to avoid race conditions on the contents of a mailbox.

The AmbientTalk kernel is used for conducting experiments in AmOP language design by relying on the following four provisions:

Mailbox Operations to add, delete and iterate over the content of a mailbox.

Mailbox Observers are first-class functions that will be invoked every time a message is added to the mailbox.

A Metaobject Protocol that allows the programmer to override the default handling of message reception and message sending [13].

Syntax Extensions can be made using a fairly simple Scheme-like macro facility, further detailed in [11].

A plethora of tentative AmOP language constructs have been reflectively implemented in AmbientTalk. Due to space limitations, it is impossible to present them all in this paper. Examples include adaptations to the communication primitives (i.e. introducing future-type message passing [45]), redundant communication primitives which route messages via different paths, group communication abstractions such as M2MI's omnihandles [26] and synchronisation primitives such as guards, chords [3] and behaviour sets [24]. The implementation of some of them is explained in-depth in [13]. More exotic language constructs such as dSelf's distributed delegation [38], which allows prototype objects to delegate to a parent located on a remote device, have been implemented as well. For the details, we refer to [41]. In the following section, one such language construct is used to conduct a realistic experiment.

5. AN EXPERIMENT: AMBIENTCHAT

This section describes an experiment that shows how AmbientTalk reconciles the expressive power of a high-level language with the ability to cope with the difficulties engendered by the hardware phenomena described in section 2.1. The experiment involves the implementation of what we call *ambient references* which can be thought of as a kind of network reference which is stable w.r.t. temporarily broken

[1]Notice that a copy of the mailbox is no longer causally connected with its owner.

network connections. This concept was subsequently used to implement a peer-to-peer instant messaging client that runs on a portable smartphone[2]. As a result, users participating in a chat session can temporarily leave one another's communication range without interrupting the session. At the application level, the code does not have to deal with this phenomenon.

The essence of the instant messenger actor is shown in the following code excerpt. As can be seen from the code, AmbientTalk uses a fairly conventional syntax. The token `::` is used to declare a public immutable slot, `:` is used to declare a private mutable slot and `:=` is used for assignment. The conventional dot-operator is used for synchronous message sending to ordinary objects whereas the `<-` operator is used to send asynchronous messages to actors. The code below describes the prototypical behaviour of an actor which can be cloned by sending it the message `new`. The exemplar actor defines a private slot to store its ID and another for the messenger's buddylist. Since a buddy is designated by its unique device-independent ID (e.g. a nickname), this is a hashtable mapping IDs to remote actors. The messenger actor allows a user to add buddies and to send text messages to a particular buddy. Calling these two methods is the responsibility of the user interface and is beyond the scope of the example.

```
InstantMessenger :: actor({
  buddies : void;
  identity : void;
  cloning.new(id)  :: {
    buddies := Hashtable.new();
    identity := id;
    provided.add(id)
  };
  addBuddy(buddyId) :: {
    buddies.put(buddyId,AmbientRef<-new(buddyId))};
  sendMessageTo(buddyId,text) :: {
    buddies.get(buddyId)<-receive(identity,text) };
  receive(from, text) :: {display(from,": ",text)}
});
```

When calling the `addBuddy` method given the ID of a buddy, an ambient reference is constructed (by calling `AmbientRef<-new(buddyId)`[3]). This ambient reference locally represents the buddy, and will be used to discover the remote actor whenever it becomes available. Furthermore, the ambient reference is resilient to the effects of broken network links, and will attempt to reconnect when possible. The `InstantMessenger` uses the ambient reference to send a text message to the designated buddy in the `sendMessageTo` method.

The implementation of the `AmbientRef` abstraction is shown below and optimally exploits AmbientTalk's reified communication traces. An ambient reference is an actor which uses AmbientTalk's discovery mechanism to find a remote actor matching a given descriptive tag. The ambient reference will essentially become a proxy to the first actor it encounters that matches the description. It is stored in the `ref` variable whose value toggles between the matching actor and `void` whenever the device containing the matching actor moves in and out of communication range. An ambient reference is constructed by sending `new` which will clone the actor below. At construction time, the clone is initialised by installing three observers on its built-in mailboxes. These will be triggered when a message arrives, and whenever its `joined` and `disjoined` mailboxes change.

Upon reception of a message, the change in the `in` mailbox triggers the `onReceive` observer. It reacts by forwarding the message to the remote actor if it is available (i.e. `ref` is not `void`). Message forwarding is implemented by changing the recipient of the message to the remote actor, and by moving it from the `in` mailbox to the `out` mailbox. When the device containing the remote actor joins (resp. disjoins) the device on which the ambient reference is running, the `onJoin` (resp. `onDisjoin`) observers are triggered. These methods take care of toggling the `ref` variable. Moreover, `onJoin` will flush all unsent messages in the `in` mailbox by forwarding them and `onDisjoin` will ensure that messages that were not transmitted yet are accumulated in the `in` mailbox of the reference in order to make sure they will be resent after rejoining.

```
AmbientRef :: actor({
  ref : void;
  cloning.new(tag) :: {
    required.add(tag);
    in.addObserver(this.onReceive);
    joined.addObserver(this.onJoin);
    disjoined.addObserver(this.onDisjoin)
  };
  onReceive(msg) :: {
    if(not(is_void(ref)), {
      out.add(msg.setReceiver(ref));
      in.delete(msg)
    })
  };
  onJoin(resolution) :: {
    if(is_void(ref), {
      ref := provider(resolution);
      in.asVector().iterate({
        out.add(element.setReceiver(ref));
        in.delete(element)
      })
    });
    joined.delete(resolution)
  };
  onDisjoin(resolution) :: {
    if(provider(resolution) == ref, {
      ref := void;
      out.asVector().iterate({
        out.delete(element);
        in.add(element)
      })
    });
    disjoined.delete(resolution)
  }
});
```

The instant messenger illustrates that AmbientTalk programs can be simple, even though they are based on a complex concurrency model and are deployed in very volatile,

[2]The AmbientTalk virtual machine is developed in pure Java, and runs on QTek 9090 smartphones on a conventional J2ME platform.

[3]This asynchronous message returns a future that is finally resolved with a value. It requires the "futures-type message passing" reflective extension of AmbientTalk mentioned in section 4.3 to be loaded.

dynamic hardware constellations. This is possible because of the right kind of language constructs, abstracting away from the more "low-level" operations that directly manipulate an actor's mailboxes. Although the abstractions seem very powerful, the real power of the model lies in the mailboxes, which directly support the cornerstones of AmOP: queuing messages to support non-blocking communication, reifying communication traces and acquainting actors based on descriptive tags.

6. CONCLUSION AND FUTURE WORK

We have defined the ambient-oriented programming (AmOP) paradigm as a set of programming language characteristics that directly deal with the hardware phenomena encountered when developing applications for mobile networks. The current state of the art of neither distributed languages nor middleware addresses all the problems resulting from these hardware phenomena. A reflectively extensible kernel language AmbientTalk was presented that fits within this AmOP paradigm. The essence of the language consist of an active object model that is based on concurrent distributed prototypes which are further equipped with eight mailboxes that constantly reflect the active object's computational state as well as the state of the hardware surrounding it.

We have implemented a number of tentative AmOP language features to illustrate the power of AmbientTalk and the AmOP paradigm it supports. Future work encompasses additional experimentation with advanced language features in order to unveil the design space spanned by AmOP. Apart from this, more insight is required on how to map AmbientTalk's principles on efficient implementation technology. More concretely, we are looking for new distributed memory management techniques, because existing techniques are not intended for use in partially (dis)connected networks. We believe that, in some cases, language constructs will need to be provided to guide the garbage collector in cleaning up lost network references because in the context of these networks, application level knowledge can probably be used to help determining those references that are no longer reachable. It is clear that a lot of the territory pertaining to language feature design for AmOP remains uncovered. With this paper we hope to promote a new branch of distributed languages, which address the problems of mobile computing from the ground up.

7. REFERENCES

[1] Agha, G. *Actors—A Model of Concurrent Computation for Distributed Systems*. MIT Press, 1986.

[2] Agha, G., and Hewitt, C. Concurrent programming using actors. In *Object-Oriented Concurrent Programming*, A. Yonezawa and M. Tokoro, Eds., Computer Systems Series. The MIT Press: Cambridge, MA, USA, 1988, pp. 37–53.

[3] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for C#. In *Proceedings of ECOOP02, volume 2374 of LNCS* (2002), B. Magnusson, Ed., Springer, pp. 415–440.

[4] Briot, J.-P., Guerraoui, R., and Löhr, K.-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys 30*, 3 (September 1998), 291–329.

[5] Callsen, C. J., and Agha, G. Open heterogeneous computing in actorspace. *J. Parallel Distrib. Comput. 21*, 3 (1994), 289–300.

[6] Caporuscio, M., Carzaniga, A., and Wolf, A. L. Design and evaluation of a support service for mobile, wireless publish/subscribe applications. *IEEE Trans. Software Engineering 29*, 12 (dec 2003), 1059–1071.

[7] Cardelli, L. A language with distributed scope. In *Conference Record of POPL '95: 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, Calif.* (New York, NY, 1995), pp. 286–297.

[8] Cugola, G., and Jacobsen, H.-A. Using publish/subscribe middleware for mobile systems. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 25–33.

[9] Cugola, G., Nitto, E. D., and Pico, G. P. Content-based dispatching in a mobile environment. In *Proceedings of WSDAAL* (2000), ACM Press.

[10] Davies, N., Friday, A., Wade, S. P., and Blair, G. S. An asynchronous distributed systems platform for heterogeneous environments. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications* (1998), ACM Press, pp. 66–73.

[11] De Meuter, W., D'Hondt, T., and Dedecker, J. Pico: Scheme for mere mortals. In *1st European Lisp and Scheme Workshop, Oslo, Norway* (2004).

[12] Dedecker, J., and Van Belle, W. Actors for mobile ad-hoc networks. In *Embedded and Ubiquitous Computing* (2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds., vol. 3207 of *Lecture Notes in Computer Science*, Springer, pp. 482–494. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

[13] Dedecker, J., Van Cutsem, T., Mostinckx, S., De Meuter, W., and D'Hondt, T. Ambienttalk: A Small Reflective Kernel for Programming Mobile Network Applications. Tech. rep., Vrije Universiteit Brussel, 2005.

[14] Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (2003), 114–131.

[15] Fjellheim, T., Milliner, S., Dumas, M., and Elms, K. The 3dma middleware for mobile applications. In *Embedded and Ubiquitous Computing* (2004), L. Yang, M. Guo, G. Gao, and N. Jha, Eds., vol. 3207 of *Lecture Notes in Computer Science*, Springer, pp. 312–323. Embedded and Ubiquitous Computing, International Conference EUC 2004, Aizu-Wakamatsu City, Japan, August 25-27, 2004.

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

[17] Gong, L. JXTA for J2ME extending the reach of wireless with JXTA technology. Tech. rep., SUN Microsystems, http://www.jxta.org/project/www/docs/JXTA4J2ME.pdf, 2002.

[18] Haahr, M., Cunningham, R., and Cahill, V. Supporting CORBA applications in a mobile environment. In *Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom-99)* (N.Y., Aug. 15–20 1999), ACM Press, pp. 36–47.

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

[20] Hewitt, C. E. Viewing control structures as pattern of passing messages. *Artificial Intelligence: An International Journal 8*, 3 (June 1977), 323–364.

[21] ISTAG. Ambient intelligence: from vision to reality, September 2003. Draft report.

[22] Joseph, A., Tauber, J., and Kaashoek, F. Mobile computing with the rover toolkit. *IEEE Transactions on Computers 46*, 3 (Mar. 1997), 337–352.

[23] 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.

[24] Kafura, D. Act++: building a concurrent C++ with actors. *Journal of Object-Oriented Programming 3*, 1 (1990), 25–37.

[25] Kahn, K. M., and Saraswat, V. A. Actors as a special case of concurrent constraint programming. In *Proc. of the OOPSLA/ECOOP-90: Conference on Object-Oriented Programming: Systems* (Languages, and Applications / European Conference on Object-Oriented Programming, Ottawa, Canada, 1990), pp. 57–66.

[26] Kaminsky, A., and Bischof, H.-P. Many-to-many invocation: A new object oriented paradigm for ad hoc collaborative systems. *17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)* (2002).

[27] Liskov, B. Distributed programming in argus. In *Distributed Computing Systems: Concepts and Structures*, A. L. Ananda and B. Srinivasan, Eds. IEEE Computer Society Press, Los Alamos, CA, 1992, pp. 370–382.

[28] 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.

[29] Mamei, M., and Zambonelli, F. Programming pervasive and mobile computing applications with tota middleware. In *Embedded and Ubiquitous Computing* (2004), IEEE, pp. 263–??? Second IEEE International Conference on Pervasive Computing and Communications (PerCom'04), Orlando (FL), U.S.A., March, 2004.

[30] Mascolo, C., Capra, L., and Emmerich, W. Mobile computing middleware. In *Advanced lectures on networking*, vol. 2497. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[31] Miller, M. The E programming language, the secure distributed pure-object platform and p2p scripting language for writing capability-based smart contracts. http://www.erights.org.

[32] Murphy, A. L., Picco, G. P., 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, p. 524.

[33] Reynolds, P., and Brangeon, R. DOLMEN - service machine development for an open long-term mobile and fixed network environment, Feb. 19 1999.

[34] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput. 39*, 4 (1990), 447–459.

[35] Schill, A., Bellmann, B., Bohmak, W., and Kummel, S. System support for mobile distributed applications. In *SDNE '95: Proceedings of the 2nd International Workshop on Services in Distributed and Networked Environments* (1995), IEEE Computer Society, p. 124.

[36] Taura, K., Matsuoka, S., and Yonezawa, A. ABCL/f: A future-based polymorphic typed concurrent object-oriented language - its design and implementation. In *Proceedings of DIMACS '94 Workshop* (1994), G. E. Blelloch, K. M. Chandy, and S. Jagannathan, Eds., vol. 18. Specification of Parallel Algorithms of *Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, pp. 275–291.

[37] Terry, D. B., Petersen, K., Spreitzer, M. J., and Theimer, M. M. The case for non-transparent replication: Examples from Bayou. *IEEE Data Engineering Bulletin 21*, 4 (dec 1998), 12–20.

[38] Tolksdorf, R., and Knubben, K. Programming distributed systems with the delegation-based object-oriented language dSelf. In *Proceedings of the 2002 ACM symposium on Applied computing* (2002), ACM Press, pp. 927–931.

[39] Ungar, D., Chambers, C., Chang, B.-W., and Hölzle, U. Organizing programs without classes. *Lisp Symb. Comput. 4*, 3 (1991), 223–242.

[40] Ungar, D., and Smith, R. B. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1987), ACM Press, pp. 227–242.

[41] Van Cutsem, T., De Meuter, W., Mostinckx, S., Dedecker, J., and D'Hondt, T. Distributed Proxies as Delegation-based Descendants. Tech. rep., Vrije Universiteit Brussel, 2005.

[42] Varela, C., and Agha, G. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices 36*, 12 (2001), 20–34.

[43] Varela, C. A., and Agha, G. A. What after java? from objects to actors. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7* (1998), Elsevier Science Publishers B. V., pp. 573–577.

[44] Weiser, M. The computer for the 21st century. *Scientific American 265*, 3 (1991), 66–75.

[45] 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.

[46] Zachariadis, S., Capra, L., Mascolo, C., and Emmerich, W. XMIDDLE: information sharing middleware for a mobile environment. In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)* (New York, May 19–25 2002), ACM Press, pp. 712–712.