A Meta-level Architecture for Ambient-aware Objects

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

Programming Technology Laboratory
Vrije Universiteit Brussel
Department of Computer Science
Pleinlaan 2 - 1050 Brussels - Belgium
{tvcutsem,jededeck,smostinc,wdmeuter}@vub.ac.be

Abstract. Programs written to be deployed in pervasive wireless ad hoc networks are notoriously difficult to program. We focus on programming languages specially tailored to writing such programs, in order to mitigate the technical constraints imposed by the ambient hardware. We describe the ramifications of the hardware properties on the software and argue in favour of a language distribution model that is aware of these effects. We conjecture that an *open implementation* of the language is necessary: a proper meta-level architecture that allows for customisation of the language to adapt it to changes in the hardware environment.

1 Introduction

Looking back at the past few decades, we observe that hardware has evolved from terminals connected to large mainframes towards ever smaller computing devices, gradually making Weiser's vision of ubiquitous computing [Wei91] and the IST Advisory Group's AmI scenarios [Gro03] feel less and less utopian. Today, developers have to weather a hardware climate of pocket-size PDAs, smartphones or cell phones, interconnected via ad hoc wireless networking technology (such as Bluetooth and WiFi) or large-scale infrastructure (such as the Global System for Mobile Communications).

Up to this day, such hardware is still programmed with languages that are inadequate for the problem domain, such as C. Even more modern languages such as Java and C# are not inherently equipped with a vocabulary geared towards distributed systems. To compensate for this, *middleware* in the form of libraries, preprocessors or component systems are the typical answer. We conjecture that distributed applications would be far easier to develop, understand and maintain if they were written in an inherently concurrent, distributed programming language. We take a bottom-up approach in designing and engineering such a language: from a characterisation of the hardware environment in which AmI programs will typically run, we distill an object-oriented distribution model with properties which we feel are fundamental to the paradigm.

Research in middleware for mobile computing has already revealed the need for application-specific customisations to the middleware. The raison d'être for such customisations is directly related to the high dynamism of the "hardware cloud": variations

* 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)

in resource availability, network connectivity and idiosyncratic computing platforms can greatly impact the performance and behaviour of applications. This has lead to the development of so-called *reflective middleware* [KCBC02, CBM+02]. We pursue this idea in a language-oriented setting by opening up the implementation of the distribution model in the form of a meta-level architecture.

We argue that there is need for an open, extensible distribution model suitable for pervasive mobile computing hardware. Therefore, we first describe the properties exhibited by the hardware in section 2, propose a programming language distribution model in section 3 and a suitable meta-level architecture in section 4.

2 Hardware Characteristics

We conceive a pervasive hardware constellation as consisting of a wireless network of interconnected devices, which could range from ordinary desktop computers over PDAs to embedded computers in intelligent refrigerators or wristwatches. Some of these devices are mobile and are constantly carried around by users, engendering what is known as *mobile computing* [Car99, MCE02]. This hardware environment is characterised by the following properties:

Volatile Connections Wireless connections are inherently less stable than their wired counterparts. This is even more so when considering mobile devices, which can roam freely and thus directly influence the quality of the network connection. Communicating programs can more easily and frequently become disconnected. Because of the prevalence of broken connections, standard exception handling techniques are no longer adequate: disconnections have become commonplace, they are no longer exceptions and should as such not be treated as one.

Ambient Resources Every device in the environment encodes its functionality in a number of components, some of which are made available as services to remote parties. To a mobile device, the availability of a service usually depends on its context (including e.g. its physical location). For example, a PDA may detect a printing service only when in close proximity to a printer. As the context changes, so will the available resources. Addressing these resources becomes more difficult. In contrast to closed stationary networks, it is usually not known beforehand which device hosts which particular service. An appropriate context-aware addressing scheme is required.

No Presumed Infrastructure Computing devices in an ambient context often function autonomously. In order to uphold their autonomy, devices should be able to directly communicate with one another, without reliance on an intermediary server. Dependencies upon servers must be minimised, as they increase vulnerability to network failures. If a client-server architecture were used, client devices would be unable to collaborate once the server has become unreachable, even though they are still in communication range. Hence, communication between devices in a wireless network should be supported by a peer-to-peer infrastructure.

These hardware phenomena have a profound effect on software written for ubiquitous hardware. They cannot in general be hidden from the application by a programming language. A language can, however, foresee these issues and offer the programmer

a specialised metaphorical toolbox with which he can build the necessary abstractions. The following section describes one such experimental toolbox.

3 An Ambient-aware Distribution Model

We consider a distribution model tailored to the construction of ambient programs. This model forms the basis of an experimental programming language we have designed, called AmbientTalk. The model is based on an extension of Hewitt and Agha's actor model of computation [Agh86], called the ambient actor model [DV04]. This model acts as the base level for the meta-level architecture described in the following section.

The unit of distribution in our model is an "ambient-aware object". We model such objects to be close relatives to actors. The actor model is a natural concurrent extension to object-orientation and has been recurrently employed in the past as the foundation for object-oriented concurrent and distributed languages. Examples are ABCL/1 [YBS86], ACT1 [Lie87] and Salsa [VA01]. An actor can be concisely described as an object with its own computing capabilities, able to send asynchronous messages to other actors. To support such asynchrony, an actor encapsulates an incoming and outgoing message queue. For the purposes of this paper, we will regard an ambient-aware object as a basic actor plus some additional properties, which will be introduced when appropriate.

We cover two aspects of the distribution model in a nutshell. In order for two ambient-aware objects on different devices to communicate, they first need to get acquainted, which requires an appropriate service discovery mechanism. Once they have a reference to one another, they can communicate given the proper communication primitives offered by the model. Other aspects of the model, such as conditional synchronisation, exception handling, etc. are outside the scope of this paper.

3.1 Context-aware Service Discovery

As already mentioned in section 2, ambient resources require the use of a sophisticated, context-aware addressing scheme. Moreover, such a scheme should not depend on predetermined name servers, as peer-to-peer communication should not be ruled out a priori. Instead of making one actor get acquainted with other actors through actor addresses, we propose language features that allow actors to "advertise" themselves using a more intensional description of the services they provide. In our model, actors can export a *provided* interface to the ambient. Actors can discover one another by announcing a *required* interface description. The process of finding acquaintances is left to the language runtime, raising the process of service discovery to a more declarative level. Consider an example adapted from [KB02] where a printing service exports a certain provided interface:

```
printer {
  properties: { dpi = 600; pageSize = "A4"; ... };
  services:
    print(document,settings); }
```

An actor that wants to address (i.e. get a reference to) a printing service can announce that it requires a service matching a certain interface:

```
discover printer where printer.pageSize = "A4";
```

When the system detects a match between the required and provided interfaces, the requiring actor is notified and passed a reference to the actor providing the requested service. Whenever the system detects that a provider is no longer available, it also notifies the depending actor. Using such a discovery mechanism, the actor can acquire a local view on the environment, being able to track status changes of only those services in which it is interested.

3.2 Asynchronous Communication

Because the connections between devices are volatile and relatively slow, we have chosen the model's communication primitives to be asynchronous and non-blocking. This also benefits the autonomy of devices: when a communication partner is temporarily unavailable, a device does not block waiting for the communication to be restored, allowing it to continue providing its services to other devices. Blocking communication would jeopardise the responsiveness of the system [MCE02].

Asynchronous communication has very distinct advantages in pervasive wireless networks. First, an asynchronous send primitive allows the sender of a message to overlap computation with message transmission, better hiding network latency. Second, when outgoing messages are queued (i.e. a send operation is not obliged to deliver the message instantly), sender and receiver are decoupled in time: the receiver of the message does not have to be on-line at the time the sender sends it. The message can be properly transmitted at some later point in time, when a connection is available [DV04].

Many a distributed language or middleware platform uses an asynchronous send operation, but the corresponding receive operation is usually blocking. This is the case in e.g. languages making use of futures such as Multilisp [Hal85] or Argus [Lis88], or in tuple space-based middleware [MCE02]. In order to maximise the availability of services and device autonomy, our model only introduces non-blocking receive operations. In fact, the receive operation is implicit. It is implemented by selecting messages from the incoming message queue for which a method is specified in the actor's behaviour.

4 An Ambient-aware Meta-level Architecture

We now turn to the description of an open implementation of the model concisely described above. Recall from the introduction that a reflective language is desirable because of the many dynamic changes exhibited by the hardware constellation.

A birds-eye view of the meta-level architecture is presented in figure 1. We will detail each of the meta-level components and describe the benefits of reifying them.

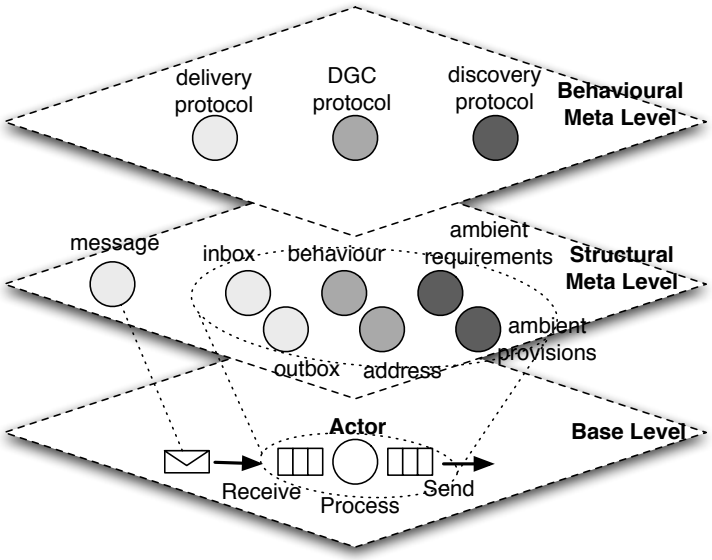

Fig. 1. Meta-level Architecture for an Ambient-aware Language

Structural Meta-level At the structural meta-level, the structure of messages and actors is reified into first-class values. An actor is decomposed into a number of objects, each responsible for different aspects of the actor's behaviour. This decomposition of one base-level value into a number of interacting meta-level values is based on similar decompositions in other meta-level architectures such as CodA [McA95] and AL-1/D [OIT92].

Inbox and Outbox The inbox and outbox represent the incoming and outgoing message queue of an actor. The reification of message queues as metaobjects allows a programmer to add messages to and remove messages from both message queues. One can also iterate over the content of the queue and register observers which are notified of state changes in the queue. This allows an actor to intervene when e.g. messages arrive in the incoming or outgoing message queue. For example, an actor can inspect messages in its inbox and discard some even before processing them. It may for example only want to process messages sent by a particular other actor. It also becomes possible to influence the order of the message queue. Rather than the default queue's FIFO order, messages can be rearranged according to some priority metric. Monitoring the outbox for messages to be sent is also useful. It allows an actor to transparently attach meta-information to outbound messages, such as network statistics or authentication information for messages sent to an actor requiring the necessary credentials.

Address and Behaviour The address is a reification of the identity of an actor. The behaviour is a dictionary of methods used to process the incoming messages. Reifying

this behaviour allows for typical introspection (e.g. inspecting an actor's methods and fields) and intercession (e.g. intervening in state updates or method invocations).

Requirements and Provisions The ambient requirements and provisions are structural reifications of the provided interface of the actor and the required interfaces of actors to be discovered in the ambient. These metaobjects are used by the service discovery protocol.

Messages A message reifies its receiver, its selector and its arguments. Moreover, a message can contain *attachments*, which is metadata or -code that is piggybacked along with the message during transmission. Attachments can influence the way the message is processed by the receiving actor.

As a concrete example, consider future-type message passing where an asynchronous message send returns a future, a handle to the result to be computed. The default evaluation of an asynchronous send returns `void`, so default messages have no knowledge of a "future address" or "reply destination" to which to return a value. The meta-architecture easily allows the extension of the language with future-type message passing, by adapting the message delivery protocol and attaching to a message a reference to the future it has to fulfill.

Behavioural Meta-level The behavioural meta-level depicts the *metaobject protocols* [KRB91] used by the language runtime to effectuate the computation. These protocols "crosscut" the structural meta-level: a protocol is executed through an orchestrated chain of message sends between structural metalevel objects. It can be adapted by overriding methods on metaobjects.

Message Delivery The language runtime delivers messages by transferring them from the outbox of the sender to the inbox of the receiver. By providing an open implementation of the message delivery protocol, different quality of service constraints can be attributed to the delivery guarantees. The default policy of the language is to always try and deliver messages. Hence, messages are kept in the outgoing queue until they can be transmitted. On resource-constrained devices or for certain kinds of messages, such as repeatedly broadcast status update messages, such guarantees may be too resource-consuming. Because the delivery algorithm is opened up via a metaobject protocol, it becomes possible to change the default delivery policy.

An adapted policy may e.g. attach a timeout period to a message. A modified delivery algorithm could then discard the message when the timeout period has passed, possibly notifying the actor who sent the message of the failed delivery. Another example is the redirection of messages from one receiver to another. Consider a message sent to an actor offering a particular service. If this service is currently offline, but it is detected that other actors are online which also offer the required service, the delivery algorithm can be enhanced to change the receiver address of the message waiting for delivery in the outbox, such that it is transparently rerouted.

Service Discovery The language runtime is responsible for matching required with provided interfaces of the different actors present in the environment. To this end, it needs to keep track of which interfaces are available on which devices. It then tests the provided interfaces against the required interfaces for which a discovery

request was issued by one of the local actors. The open discovery protocol allows actors to intervene in this matching process by providing their own comparator functions.

Custom comparators can be useful for resolving interface versioning problems. When different versions of an interface exist, an actor may fail to find another actor in the network which provides a different version of the required interface. By plugging in a comparator with knowledge of the different versions, the interfaces could still be matched successfully. Note that in this case, the discovery algorithm must also be adapted to not hand the requiring actor a reference to the real provider, as this actor does not provide the presumed interface version. Rather, it must be handed a proxy which implements the required interface version and which converts this interface into the provided interface version.

Another aspect of the discovery protocol is the detection of services that have become unavailable, such that actors depending on that service can be notified. To detect unavailability of devices in general, a heuristic based on timeouts is used. An application may, however, define more suitable application-specific heuristics. The protocol can e.g. be adapted to signal the unavailability of services located on remote devices, even when these devices remain online. Some programs may e.g. explicitly want to make services unavailable to save battery power or due to load balancing issues. Conversely, one may want to inhibit the protocol from notifying actors of the unavailability of a device. If the same service is offered by different devices in the environment and one of them disconnects, it would be more appropriate for the discovery protocol to transparently reconnect actors depending on that service to a similar service on another device, rather than signalling them that the service is unavailable.

Distributed Garbage Collection Distributed garbage collection of actors in open networks is notoriously difficult to automate. Wireless networks surrounding mobile devices easily become partitioned and actors may acquire references to remote actors which move out of range and are never encountered again. The problem stems from the fact that broken references are not necessarily treated as exceptions. A garbage collector cannot detect whether a broken remote reference will be mended at some point in the future, making it impossible to perform fully automatic garbage collection. On the other hand, we want to stay clear from manual memory management, as it would introduce an extra concern for the application programmer.

Our proposed solution is to allow the programmer to “guide” the garbage collector by hinting at which references may be collected. Leasing protocols, where remote references expire unless they are explicitly renewed, are one abstraction mechanism which suits this approach. By referencing a remote object through such a “lease”, the programmer allows the reference to be collected if the lease is not renewed before time.

Our experimental language AmbientTalk, which implements the distribution model outlined in section 3, partially implements the above meta-level architecture and allows access to message queues, actor behaviour and required and provided interfaces. The protocols described above are currently only implemented implicitly and are inaccessible to the programmer. We plan to open up the implementation further such that

the language extensions described above become possible to implement. Future work also entails using more principled reification mechanisms, such as mirrors [BU04], to represent metaobjects and to effect the shift from base to meta-level.

5 Position Statement

We approach Ambient Intelligence from the point of view of wireless, pervasive networks inhabited by mobile devices. Having pinpointed the harsh characteristics of the hardware, we find that present-day programming languages have no inherent language features to comfortably abstract from them. We do not consider intricate assemblies of `if` and `try-catch` statements hidden underneath thin library or middleware interfaces to be comfortable abstractions. Neither are we proponents of a distribution-transparent approach, where the nature of abstractions is such that the programmer is no longer aware of the dangers of distributed programming, viz. partial failures and synchronisation. Some issues must inevitably percolate into the application layer.

We take a language-oriented path and conjecture that a language with an innate knowledge of concurrency and distribution issues eases development. We have proposed a model where this knowledge is embodied in actor-actor and actor-ambient communication primitives. Moreover, the concept of an open implementation is critical to distributed programming languages targeted at dynamic hardware constellations. It allows the programmer to deal with unforeseen situations without having to adapt his entire program. Using a metaobject protocol, the programmer can change the concepts of the language to better suit the application, rather than having to change the application to better suit the imposed language concepts.

References

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[BU04] Gilad Bracha and David Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the 19th annual Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 331–343, 2004.

[Car99] Luca Cardelli. Abstractions for mobile computations. In *Secure Internet programming: security issues for mobile and distributed objects*, pages 51–94. Springer-Verlag, London, UK, 1999.

[CBM+02] Licia Capra, Gordon S. Blair, Cecilia Mascolo, Wolfgang Emmerich, and Paul Grace. Exploiting reflection in mobile computing middleware. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):34–44, 2002.

[DV04] J. Dedecker and W. Van Belle. Actors for Mobile Ad-hoc Networks. In L. Yang, M. Guo, G. Gao, and N. Jha, editors, *International Conference on Embedded and Ubiquitous Computing EUC2004*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, August 2004.

[Gro03] IST Advisory Group. Ambient intelligence: from vision to reality, September 2003. Draft consolidated report.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.

[KB02] Alan Kaminsky and Hans-Peter Bischof. 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*, pages 72–73, New York, NY, USA, 2002. ACM Press.

[KCBC02] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Commun. ACM*, 45(6):33–38, 2002.

[KRB91] Gregor Kiczales, Jim Des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[Lie87] Henry Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications Of The ACM*, 31(3):300–312, 1988.

[McA95] Jeff McAffer. Meta-level programming with coda. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 190–214, London, UK, 1995. Springer-Verlag.

[MCE02] Cecilia Mascolo, Licia Capra, and Wolfgang Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag New York, Inc., 2002.

[OIT92] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the Workshop on New Models for Software Architecture*, November 1992.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36(12):20–34, 2001.

[Wei91] M. Weiser. The computer for the twenty-first century. *Scientific American*, pages 94–100, september 1991.

[YBS86] Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.