Actors for Pervasive Computing

Jessie Dedecker*and Wolfgang De Meuter and Werner Van Belle
(jededeck—wdmeuter—we47091@vub.ac.be)
Tel: +32-2-629.35.30 - Fax: +32-2-629.35.25
Vrije Universiteit Brussel
PROG - Department of Informatics
Pleinlaan 2
1050 Brussels
Belgium

1 Introduction

The position we want to defend in this paper is twofolded:

1. First, we take the position that asynchronous communication patterns are to be considered as the main communication paradigm between pervasive computing devices.

2. Second, we argue that the actor model, which employs asynchronous message passing as its main communication paradigm, can be extended to provide better support for pervasive applications.

Pervasive or ubiquitous computing means that the technology should be so gracefully integrated in our everyday life that the user is not aware of the technology anymore [Wei91]. A good example of this is wearable computers. To achieve such a goal the software should adapt each time a user moves to another environment. The act of moving is part of almost anybody's everyday life. This means that pervasive applications should be able to detect and communicate with other devices that they encounter in their everly changing environment. Because the user moves from one space to another connections are lost and new connections are created. This makes communication between pervasive applications very volatile. Communication over such volatile networks is difficult by means of traditional methods (such as RPC [BN83], RMI [WRW96] and other synchronous variants). With synchronous communication solutions there is an assumption that the communication partner on the other end is available. A communication partner that is not available is the exception rather than the rule. This is a glaring contrast to the frequent movement of users that move from one space to another. For this reason we find that existing synchronous communication models (such as the ones mentioned above) are not appropriate for pervasive communication. In this position paper we advocate the use of appropriate language constructs to support asynchronous communication

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

models for pervasive applications.

2 Pervasive Applications Communicate Asynchronously

Asynchronous (a.k.a. non-blocking) communication can perhaps best be compared to the act of sending a letter via snail mail or electronic mail. After somebody has sent a letter, he continues with his life and later (somewhere in the undefined future) when he receives a reply to his letter he can read it and respond accordingly. To come back to the world of computers, computation continues after sending an asynchronous message. So, there is no correlation between the time of sending and the time of receiving a message. However both communication partners must be located in the same physical space. Synchronous (a.k.a. blocking) communication, on the other hand, can best be compared with a telephone call to someone. When you call someone and you ask him a question, then you usually get the answer immediately within the same telephone call. In the world of computers it means that, with synchronous messages, communication can only occur when both communication partners (sender/receiver) are available at the same time and place.

Pervasive applications often imply a disconnected operation because there can be a very large delay between sending a message and receiving an answer. In extreme cases the delay could be several days or months, for instance when the device is not connected to any network at the moment of sending a message. Therefore, pervasive applications communicate inherently asynchronously. However, implementing applications that work asynchronously is not easy at all. Below we discuss two major problems.

2.1 Computational Context

Asynchronous communication is not a bed of all roses and skittles – it comes with a cost and that cost is the complexity of developing applications. The complexity comes from the fact that computation continues after sending a message. So, after an asynchronous message is sent the actor receives other messages. When the reply to the first message is received the program needs to restore the context in which that message was sent in order to interpret the result it receives. If we go back to our metaphor of sending a letter through e-mail we see that the context is also reminded in the reply of the message using quoted text – or – in the case of a letter sent by snail mail companies often use reference ids in their letters that should be placed on each reply to that letter. The act of storing the computational context when an asynchronous message is sent and restoring the computational context when the reply is received is very complex and leads us to some kind of state based programming.

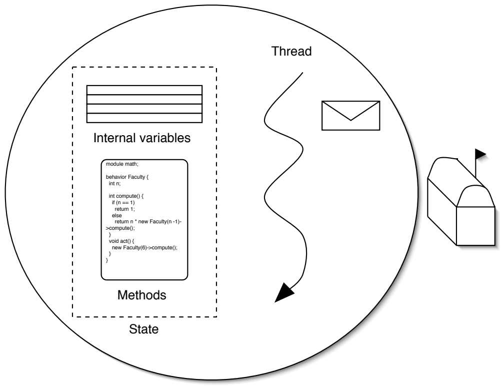
2.2 Environmental Context

Due to the continuous movement of the user from one physical space to another, the available communication partners often change. The devices will often have to postpone tasks it was doing. When the user moves again in communication range of a communication partner then it should resume its task, unless of course the task has become obsolete (i.e. because the device was able to make a booking at some other place). With physical movement the device starts living in another environment with different possibilities (other devices become available) and different limitations (devices disappear).

2.3 Actors

Actors [Agh90] are active objects that each have their own thread of execution attached to them. Messages between actors are sent asynchronously. When an actor receives a message it can [Agh90]:

- create a new actor
- send messages to known actors. The message can contain the address of other known actors
- modify its own state, there is no shared data between actors



```
module math;

behavior Faculty {
  int n;

  int compute() {
    if (n == 1)
      return 1;
    else
      return n * new Faculty(n -1)-
>compute();
  }
  void act() {
    new Faculty(6)->compute();
  }
}
```

Figure 1: Actor Model

Figure 1 (from [VA01]) depicts the internal model of such an actor. An actor is internally equiped with a mailbox where all the received messages sent to the actor are collected. An actor processes one message at a time until the mailbox is empty. When the mailbox is empty the actor waits for a message. There is no guarantee in which order messages in the mailbox get processed.

There is no support for the context switching problems associated with asynchronous communication mentioned above. In [VA01], the actor language is extended with linguistic support. The linguistic support helps expressing synchronisation and sequentiality between a set of actors. For example, syntax is added to put a sequential order on a set of messages.

2.4 Limitations of the Actor Model

The actor model has some limitations with respect to pervasive computing:

- the model does not support disconnected operation. This occurs frequently when a user is moving from one space to another. When a user moves out of range from a certain network, then messages sent by an actor are lost – and vice-versa the actors that were living in that physical space and that send a message to an actor living on the device that user was carrying are lost. In some cases, the loss of messages is unacceptable, i.e. in the case the actor was making a reservation for some restaurant in the neighbourhood. Ideally, the interaction between the actors should resume when the user enters the space again.
- the model requires the application to know its available communication partners when the user arrives at a new location. This is clearly unacceptable, because it conflicts with the preposition that the application should adapt to new environments. We need a more abstract way to reference the set of actors the application can communicate with.
- multiple communication partners are com-

mon in ad-hoc networks [lKB02], like the ones we envision for pervasive communication. For example, sometimes a pervasive computing device would like to communicate with all the devices in its neighbourhood.

- no support for computational/environmental context – the actor model lacks support for the loss of context, as explained above.

3 Changes to the Actor Model

In this section we make some preliminary propositions to extend the actor model so that the limitations, pointed out above, are alleviated.

3.1 The Outbox

The current actor model encorporates one mailbox for the incoming messages. So, when a message is sent by an actor it is sent immediately to the mailbox of the receiving actor.

We propose to extend the actor model with an outgoing mailbox (outbox) next to the incoming mailbox (inbox). When an actor sends a message it gets posted in its outbox. Messages in the outbox are then sent out to the receiving actors. When a message is undeliverable, for example because either the sending and/or the receiving actor has moved out of range of the network, then the message is kept in the outbox. At regular intervals the actor retries to send the message. The outbox is useful for disconnected operation, because it allows the actor to send messages at the time it has that intention, but when it is impossible to reach the other actors. The mechanism of an outbox takes away the burden of regularly having to check if an actor has become available for communication.

3.2 Multiplexed Inboxes and Outboxes

In the current actor model there is no order in which received messages are processed. To have better support for the different tasks an actor is performing we propose to classify incoming and outgoing messages. The incoming and outgoing messages are classified depending on their computational and environmental context. For example, in the case of environmental context, messages sent to an actor that is currently unavailable could be classified together so that when that actor becomes available, all messages are sent out at once. Another example, in the case of computational context, all messages received about the activity of making a reservation are classified all together. By classifying the incoming and outgoing messages we can reduce the burden of the developer, because the actor can process all the messages that belong to the same context sequentially.

3.3 Filters

We already explained above that we can classify incoming and outgoing messages to help with the problem of context switching. Filters are used to classify incoming and outgoing messages depending on the context they belong to. When a message is sent or received by the actor, then it must be classified. We propose to use a declarative language to classify the incoming and outgoing messages.

3.4 Explicit Mailbox Management

We foresee that there will be a need for describing some quality of service (QoS) properties when sending a message. These QoS properties can range from very simple things like how long should the actor retry sending a message before giving up to more advanced QoS properties like timing constraints, properties depending on the state of the actor. For this there will be a need to manage the inbox and outbox explicitly. More concretely we propose a management system that can:

- Add and remove messages
- Determine what message should be processed next by the actor

3.5 Reference by Query

Currently, an actor has two ways for getting a reference to another actor:

1. by name
2. passed an argument in a message

When a pervasive computing device enters a completely new space then it does not know any names of the actors living in other devices, nor do the inhabitants of that space know the new actor. We need to introduce language constructs for referencing actors in other ways than just their name. For this reason we propose a mechanism for referring to an agent using a query on the messages that the agent should understand.

We can implement such a query mechanism in three ways:

1. by creating one actor in each space with a standardized name that is serving as a receptionist and helps new immigrating actors to know the inhabiting actors of that space.
2. by sending out a broadcast message to all actors in the network. Actors receiving the message can then decide whether and how to handle this broadcast message.
3. by a combination of the two above. That is, the receptionist actor is identified using a broadcast message and is from then on used to retrieve the references to the inhabiting actors. The combination approach is also used by Jini[1] [Edw99].

The first and third method require a separate service (a receptionist actor in our case) to be available in the network and this is not always feasable. This is especially true in the case of ad-hoc networks. For example, three persons that go hiking in the forest, all wearing devices (such as GPS) to help them. All these devices form a dynamic network and there is no receptionist service available in such a dynamic environment.

4 Conclusion and Future Work

In this paper we argued that the asynchronous communication patterns are to be considered as the main communication paradigm in a pervasive computing context. Asynchronous communication patterns fit well in a pervasive computing context, because communication in ad-hoc networks is asynchronous. Mapping synchronous communication patterns, such as RPC or RMI, put lots of work on the programmer's shoulders.

We also argued that the actor model, which supports asynchronous communication patterns, offers a good starting point. However, the actor model has some limitations when applied in the context of pervasive computing. We pointed out that the actor model can be extended to provide better support for use in the context of pervasive computing.

[1]Trademark of Sun Microsystems

We are currently developing an actor language that is based on the extensions proposed in section 3. Afterwards we will implement some concrete examples to test the usability of our extensions.

References

[Agh90] Gul Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[BN83] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. In *Proceedings of the ACM Symposium on Operating System Principles*, page 3, Bretton Woods, NH, 1983. Association for Computing Machinery.

[Edw99] W. Keith Edwards. *Core Jini*. Prentice Hall PTR, 1999.

[lKB02] lan Kaminsky and Hans-Peter Bischof. 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.

[VA01] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.

[Wei91] M. Weiner. The computer for the 21st century. *Scientific American*, 265(3):66–75, 1991.

[WRW96] A. Wollrath, R. Riggs, and J. Waldo. Adistributed object model for the Java system. In *2nd Conference on Object -Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, 1996.