

## The Message-Oriented Mobility Model

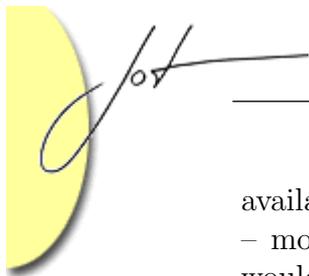
**Jorge Vallejos, Tom Van Cutsem, Elisa Gonzalez Boix,  
Stijn Mostinckx, Jessie Dedecker, Wolfgang De Meuter**  
Programming Technology Lab, Vrije Universiteit Brussel, Belgium  
{jvallejo,tvcutsem,egonzale,smostinc,jdedeck,wdmeuter}@vub.ac.be

Mobile networks composed of devices interconnected by wireless communication media frequently suffer from partitions. If mobile devices depend on software services running on remote devices, such partitions may render the software services unavailable. We propose the use of code mobility to mitigate the unavailability of software services in mobile networks. We discuss the issues of existing mobility mechanisms, identify four characteristics necessary to support code mobility in mobile networks, and propose a model for code mobility, the Message-Oriented Mobility (MOM) model, that features such characteristics.

### 1 INTRODUCTION

Currently, wireless technology is revolutionising the way software systems serve their users. New visions of computing can be realised where users are continually surrounded with mobile and embedded computing devices. Whereas these scenarios are becoming ever more realistic from a technical point of view, programming such devices remains notoriously difficult due to the limited resources and volatile connections these devices can sustain between each other. Mobile networks composed of such devices frequently suffer from partitions due to the combination of the volatile connections and the physical mobility. If the mobile device depends on software services running on remote devices, such partitions result in those services becoming unavailable.

In this work, we explore the use of code mobility to circumvent the problem of service unavailability due to network partitions in mobile networks. Software services provided with the capacity of migrating from one device to another are less dependent on individual devices – services are not constrained to always run on the same device – and are consequently less vulnerable to the effects of network partitions on the availability of these devices [4]. A mobile service can, for instance, follow its user hopping from one device to another as the user moves about and uses a new device (a scenario frequently studied in the context of mobile computing under the name of *follow-me* services [11, 3, 18, 16]). Thus, the service can either continue working on the new device or otherwise “hibernate” and be moved to an appropriate host later after having been transported by the user in a mobile device like a PDA. However, this solution only works if the network partition is predictable: the service has to be moved when the connection to the departing device using the service is still



available, e.g. on request by the user. Hence, we focus on non-automatic – *proactive* – mobility. An automatic mechanism for mobility is unsuitable because services would need to be moved automatically upon disconnection, which is impossible at that point.

In the field of programming language research, proactive code mobility schemes have been traditionally associated with a `move` instruction that expects e.g. an object and a location, and which pushes the object towards that location [20, 25]. We argue in this work that this instruction has the same implications as the `goto` instruction in the sixties. We present a list of arguments against `move` that stem from software engineering and security concerns, and formulate the need for a more structured mobility mechanism in order for programmers to be able to track the location of objects.

To achieve proactive mobility of running services in mobile networks, this paper proposes an object-oriented programming language model called the *message-oriented mobility (MOM) model*. The contributions of this work are threefold:

1. We propose a structured language mechanism that aligns mobility with message sending: objects can implement special “move methods” which, when invoked, collocate the objects with their caller. This mechanism allows a service programmer to express when and what part of a service to move while protecting the security of the devices involved in the move process.
2. We describe a programmer-transparent mechanism of strong mobility which can transfer running services between devices while taking care of stopping a service, saving its program state, and resuming it remotely.
3. We complement the move process with a decentralised network reconfiguration mechanism that automatically updates the references to the moved service over the network.

The paper is structured as follows. Section 2 identifies the requirements of proactive mobility models for mobile networks and the issues of existing code mobility models in this context. Section 3 describes the different components of the message-oriented mobility model. Section 4 evaluates our mobility model with respect to the requirements identified in this paper. Section 5 validates this model by presenting a high level mobility pattern for *follow-me* services. Before concluding, we describe related work in section 6.

## 2 MOVING SERVICES IN MOBILE NETWORKS

In this section, we identify the requirements of proactive code mobility models for mobile networks and discuss the issues of existing code mobility models in this context.



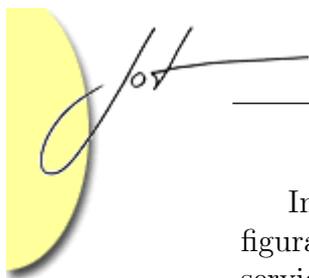
## Requirement #1: Strong Mobility

As explained in the introduction, we require a service to be moved in order to fully preserve its availability. Moving a running service is a decision that not only concerns the data of the service program, i.e. behaviour and data state, but also the computations in which the running program is involved at the time of the move. This “runtime” state is typically a combination of a heap of data objects and a stack of activation records which traces the service’s control flow and represents all unfinished computations. Both have to be moved if the runtime state of the program is to be transferred intact. The type of mobility where data and computations are moved between devices is known as *strong mobility* [8].

In the literature, a lot of mobility schemes are described where only data is moved [12]. There are multiple reasons for this choice: the runtime stack of a service is usually not reified in the language, it is not serialisable, and it is tightly bound to the execution platform. These limitations lead to so-called *semi-strong* mobility schemes requiring programmers to manually encode the runtime state of a service as data upon migration, and to decode this data via a number of branching instructions after migration to get back into the correct computational context where the program was before migration. A semi-strong mobility scheme requires programmers to foresee all possible ways in which a service can be moved. As there is no mechanism which can transparently take care of migrating the service at any point, at any time, all possible states in which a service can be moved have to be statically determined. From a user’s point of view, this means that he cannot always move a service from one device to another. However, the effect that we want to achieve is that a service can be migrated as unrestrained as possible e.g. it can be hibernated when placing a device in sleep mode. Thus, we claim there is a need for a strong mobility mechanism that migrates both data and computation and which circumvents the problems of manually serialising the runtime stack of a program.

## Requirement #2: Decentralised Network Reconfiguration

Deciding how and what part of the service should migrate from one device to another is not the only issue programmers need to tackle when implementing code mobility. The change of location of the moved service may lead to inconsistencies in the communication between this service and the rest of the services over the network, e.g. references pointing to the old location of the moved service. Early code mobility approaches, mostly developed for *load-balancing* purposes [15], avoided this problem by collocating the services that communicate often (services were moved together with all the software resources they needed to operate properly). This technique is no longer applicable, however, because some services cannot, and in some cases should not, be moved, e.g. fixed hardware resources such as screens and speakers, or software containing confidential information such as address books or personal agendas.



In the context of mobile networks, we require an extra step of network reconfiguration in the move process that ensures correct communication with the mobile service at any moment of the move. Such reconfiguration mechanism should be reliable even in the presence of network partitions, which also rules out message forwarding schemes and *global* reconfiguration strategies (realised by central servers [9]). In mobile networks, a device cannot rely on another one playing the role of higher authority since there is no certainty that connection with such an authority will continue to be available in the location where the interaction happens. We require a decentralised reconfiguration mechanism similar to the techniques used in peer-to-peer architectures [19, 17].

### Requirement #3: Structured Mobility

It is common practice to incorporate mobility in a programming language by explicitly introducing the concept of location and by complementing it with a simple `move` instruction [8, 20]. We only partially agree with that. Surely, mobile programs are written with the explicit assumption in mind that parts of the program will reside on and move between different locations. However, this does not imply that mobile programming languages should explicitly model the concept of a *location*. Dealing with absolute locations seems like an unattainable language design choice when one has to deal with mobile networks which are inherently dynamically defined.

A more fundamental problem with the `move` operator is that it bypasses the language's interaction mechanisms (such as message passing, function calling and process spawning) and moves a running process or object to a specified destination device. We argue that such an operator is suspiciously comparable to the `goto` operator and may be one of the major obstacles for code mobility to be lifted from a technical programming trick to solid engineering levels.

Similar to the `goto` instruction, a `move` instruction has a strong tendency to obscure the actual locations of objects. Especially since distributed programs have a vast arsenal of object-oriented programming techniques such as regular control flow instructions (such as `while` or `for`), late bound message passing, double dispatch, meta programming and exception handling at their disposal. With only a few lines of code that use this construct, networked object-soups can be created the structure of which cannot be predicted or understood by programmers. Analogous to the case of structured programming, we claim there is a need for structured mobility, i.e., language mechanisms that can help programmers to determine the relative loci of objects by reading the code.

### Requirement #4: Secure Mobility

The migration of a service may entail a number of security issues at the devices involved in the move process, i.e. the current and new hosting devices of the mobile



service. The migration of a local service without the acknowledgement of the current hosting device (e.g. system's or user's permission) may affect its proper functioning and cause privacy problems (e.g. when moving a database containing confidential information). This is what occurs in mobile agent technologies. In such approaches, agents can *autonomously* move to new devices (using a *move* instruction). This property is identified as a security hazard by Vigna and Thorn [20, 25] who state that mobile agents are suspiciously similar to *worms* that once launched they can be hard or impossible to control.

The arrival of the service to the new hosting device may also produce security issues which are mainly related to resource rebinding. In traditional process migration mechanisms, security problems are mitigated because there is little or no resource rebinding [15]. However, when a mobile service needs access to local file systems, windowing systems, networking interfaces and so on, security problems arise. In order to protect the resources of the devices in mobile networks, we require a mobility mechanism that enables them to autonomously decide what to expose to the moved service. As previously explained in this section, this decision cannot rely on a central server due to the connection volatility that exists between the server and the mobile devices.

We argue that a secure mobility mechanism should be conceived as a *two-party contract* between the devices involved in the move process, more specifically, the service that is going to be moved and another residing on the device receiving the moving service. Thus, services are neither pulled unilaterally by remote hosts nor pushed unilaterally to a remote host.

## Summary

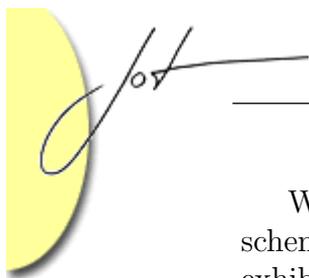
In this section, we present a number of requirements to support code mobility in mobile networks. We summarise these criteria as follows:

**Strong mobility** To preserve service availability, we require a mechanism of strong code mobility that consistently moves both the behaviour and the runtime state of a service without putting extra burden on programmers.

**Decentralised network reconfiguration** We need a decentralised network reconfiguration mechanism that automatically updates the references to the moved service over the network, and which is resilient to network partitions.

**Structured mobility** We require a code mobility mechanism that enables programmers to write structured mobile programs and to predict the relative loci of objects by reading the code.

**Secure mobility** Security should be conceived as a *two-party contract* between the services involved in the move process. Services should neither be pulled unilaterally by remote hosts nor be pushed unilaterally to a remote host.



While some of the criteria explained above can be observed in other mobility schemes or distributed languages, to the best of our knowledge no single approach exhibits all of them. We further discuss the related work in Section 4.

### 3 THE MESSAGE-ORIENTED MOBILITY (MOM) MODEL

To address the requirements discussed in the previous section, we introduce the *MOM model* for code mobility in mobile networks. It extends the actor model [1] of concurrency and distribution with the notion of *move methods* which enables programmers to work with strong code mobility in a structured and secure manner. In this section, we describe the three constituent parts of the MOM model: move methods, strong mobility of actors, and decentralised network reconfiguration.

In what follows, we employ an actor language, called AmbientTalk, to illustrate the use of the MOM model. AmbientTalk provides dedicated features for the software development of mobile services, some of which we have used to implement our code mobility model. For the sake of conciseness, we do not present an in-depth discussion of AmbientTalk itself. Instead, we introduce specific features as necessary in the course of this section and refer the reader to dedicated publications [5, 6] for more information about this language.

#### Move Methods

In the MOM model, services are represented as actors which are also the units of mobility. An actor can declare that it is allowed to be moved by other actors by implementing a new kind of method called a *move method*. An actor on a remote device can declare that it is willing to receive a mobile actor by sending it a move message, which is any message corresponding to one of the mobile actor's move methods. Upon processing this message, the mobile actor is moved from its current location to the location of the sending actor. In other words, the sender pulls the receiver actor towards its own location.

The following code sample sketches the definition of a mobile game application implemented as an actor in AmbientTalk (see more details of this implementation in Section 5):

```
makeMobileGame(userID,localServices):: actor({
  guiActor: void;
  user: void;
  // actor constructor
  init(): {
    guiActor:= localServices.get("gui");
    user:= userID
  }
})
```



```
...
// a move method
move come(services):: {
  // rebind the GUI resource
  guiActor:= services.get("gui");
  ...
}
})
```

AmbientTalk actors (declared using the `actor` construct) consist of a number of fields and methods which are referred to as the actor's *behaviour*. As shown in the example above, fields are defined using a `name : value` syntax while methods are defined using a `name(parameters) :: body` syntax. Move methods are distinguished from ordinary methods by a `move` keyword that is prefixed before the method name. In the example, `come` is a move method. An actor may implement multiple move methods, each with their own distinct name.

The variables of an actor may refer to two kinds of values: regular objects or other actors. When an actor is moved, all regular objects are moved along (i.e. they are transferred by-copy). However, variables that referred to actors at the old location will become remote references at the new location. In other words, acquaintance actors are transferred by-reference.

When one of an actor's move methods is invoked, before executing the body of the move method, the actor is first moved to the location of the object that invoked the method. Only upon arrival at that location is the body of the move method executed. Hence, the body of a move method serves to do "post-move processing". This enables the moved actor, for instance, to rebind services that were left behind at its old host to equivalent services at its new host or to pull acquaintance actors from its old location to its new location. Note that such service rebinding is performed through parameter passing: the actor initiating the move, e.g. the sender of the `come` message, explicitly passes all (references to) services it wants to share with the moving actor as arguments to the move method, e.g. a `guiActor` service, and the moved actor receives these parameters once it arrives. In the body of the move method, the moved actor can then choose whether to keep referring to the original service at its old location or to rebind the variable to point to the local service passed to it.

A final aspect of the semantics of move methods is that, when the actor to move is already colocated with the sender actor, the body of the move method is not executed (i.e. the request to move is ignored). Because the actor does not need to be moved, it is not necessary to do any post-move processing.

## Strong Mobility of Actors

The MOM model provides a strong code mobility mechanism that moves the data and computation of a running service without suffering from the problems described in Section 2 raised by having an implicit runtime stack. We benefit from the event-driven nature of actors for this purpose. Actors are structured around an event loop which responds to incoming events – modelled as messages – and handles one message at a time by dispatching to the appropriate event handler (method). Parts of an actor which have to be performed “in the future” are not stored in an implicit runtime stack, but are rather encapsulated in a message which is put into the actor’s (explicit) event queue. A service structured according to this model has the property that its runtime stack is empty every time it handles the next event in the queue. All “future” computations are stored as events in the message queue. Hence, if the request to move is modelled as an event (a message), an event-driven service can respond to such a “move” event by moving its data and its (accessible and serialisable) event queue and be confident that it has not lost any computational context. Upon arrival, the service simply continues its perpetual event loop.

The implementation of strong mobility for actors consists of moving the actor from one device to another while taking along all messages already sent to it but not yet processed by it (i.e its queue). Such messages represent the future computational context of the moving actor, so it is imperative that they are taken along. If this would not be the case, the actor would lose some of its computational context upon arrival, resulting in mere semi-strong mobility. Messages sent to an actor while it is moving should be properly received by that actor. This is made possible because message passing and message processing are decoupled in the actor paradigm: while the actor is moving to its new location, its message queue will still be available for message reception at its old location until the actor has been fully moved.

The semantics of invoking a move method can be described as follows. When the next message from an actor’s message queue is processed, method lookup is first performed in order to find a method corresponding to the message. If the method found is a move method, the implementation checks whether the actor that sent the message is already collocated with the receiver. If this is the case, the message is further ignored. Otherwise, the move process is initiated and the receiver actor is moved to the location of the sender. Upon arrival, the method body is executed, parameterised with the arguments passed to it by the sender actor. Before explaining the move process in-depth, Figure 1 shows the effect of moving an actor (and its message queue) and identifies the different parties which play a role in the process.

The move process consists of three steps which happen transparently from the point of view of the programmer. This process requires the active participation of the *initial host* of the mobile actor and the *pulling host* from which the move message is sent (see the top left quarter of Figure 1). The three steps of the move process are described in detail below:

1. The moving actor stops processing messages and the initial host asks the

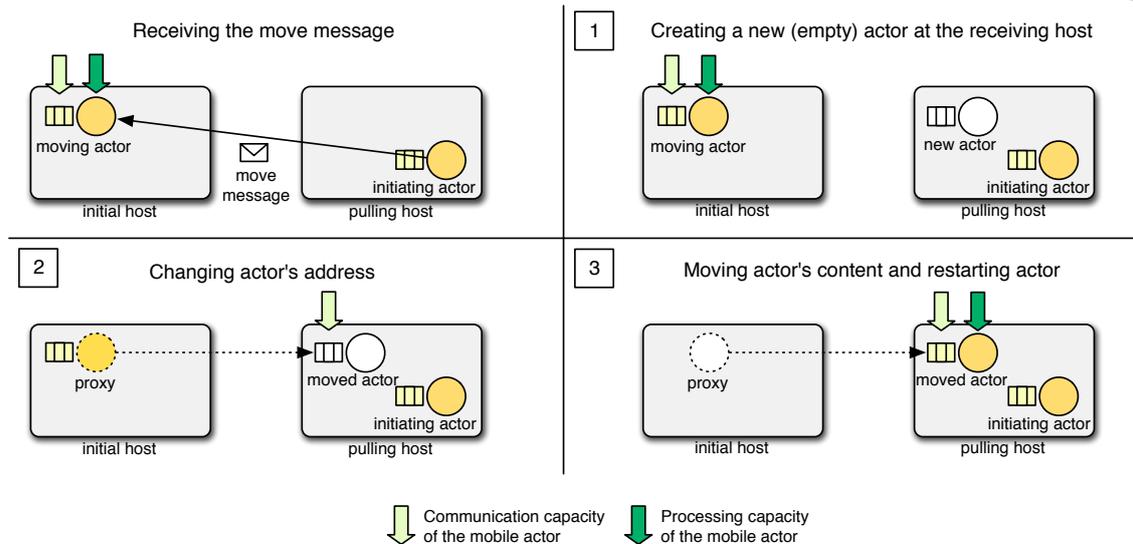


Figure 1: The move process of an actor.

pulling host to create a new actor (see the top right quarter of Figure 1). As described later on in the process, this new actor will become the moved actor at the pulling host. This new actor is created with an empty behaviour and is deactivated, it will not process any messages. Nevertheless, this actor has a message queue and hence is already capable of receiving messages.

2. The moving actor continues receiving messages at the initial host until the pulling host sends back a reference to the new, moved actor. From that moment on, the moving actor becomes a proxy and all messages sent to it will be forwarded to the moved actor (see the bottom left quarter of Figure 1). This delegation happens transparently from the point of view of other actors currently communicating with the moving actor.
3. The moving actor's behaviour and all messages in its incoming message queue are sent to the moved actor. When the behaviour and the message queue of the moving actor arrives at the pulling host, they are used to reinitialise the moved actor. The content of the old actor's message queue is placed in front of any messages which may already be present in the moved actor's message queue, because chronologically they were received by the moving actor before any message could have been received by the moved actor. After this reinitialisation, the moved actor is activated and starts processing messages (see the bottom right quarter of Figure 1).

## Decentralised Network Reconfiguration

The move process explained above requires a final network reconfiguration step to ensure proper functioning of the system once the actor has moved. It is a decentralised mechanism that performs a transparent update of references to the moved actor. Three different cases requiring network reconfiguration have to be considered after movement (see Figure 2):

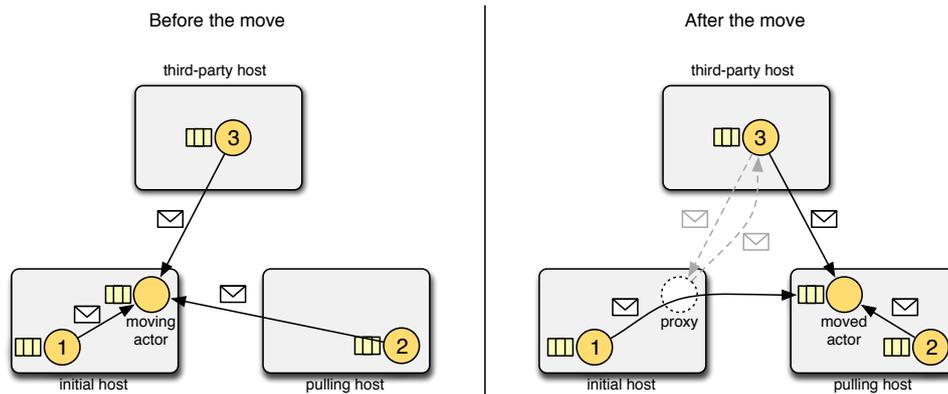
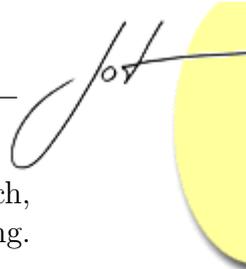


Figure 2: Decentralised network reconfiguration.

1. Actors at the initial host which locally referred to the moving actor before it moved, refer to the proxy afterwards. Messages sent to it are rerouted to the moved actor at the pulling host.
2. Actors at the pulling host which remotely referred to the moving actor before it moved, refer locally to the moved actor afterwards. Messages sent to it are passed directly to the moved actor (no roundtrip via the initial host is necessary).
3. Actors at any other third-party host remain referring to the actor at the initial host after it moved. When an actor at the third-party host sends a message to the moved actor at the initial host, this host informs the third-party host of the new location of the moved actor. The third-party host will then resend the message to the pulling host and updates its remote reference to the moved actor. Using this mechanism, chains of forwarding proxies can never be formed which improves reliability and performance.

Note that in the third case, the reconfiguration mechanism requires the active collaboration of the initial host, which is responsible for updating remote references that previously referred to the moving actor. As explained in the introduction, such collaboration is not always possible in a mobile network, as this device may become unavailable due to a network partition. In order to cope with this problem, we use



special remote references found in AmbientTalk called *ambient references* which, when broken, enter a search mode to rediscover the moved actor via multicasting. For more information about these references, we refer to [23].

## 4 DISCUSSION

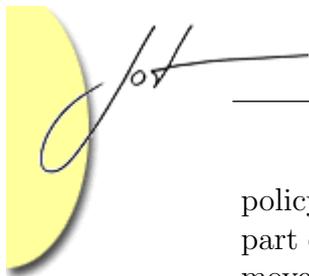
We now evaluate the MOM model in the light of the four criteria for code mobility in mobile networks identified in Section 2.

**Strong Mobility** The MOM model provides a strong mobility mechanism for actors. This mechanism enables an actor to gradually move (first its behaviour and then its runtime state which is reified as a message queue) while ensuring the availability of its services for other actors over the network. During the move process, none of its incoming messages are lost, such that the actor retains its full computational context upon arrival. It is important to recall from Section 1 that we focus on *proactive* mobility i.e. the actor is moved when the connection between its old and new locations is still available. The MOM model does not explicitly deal with disconnection during the move process.

**Decentralised network reconfiguration** The MOM model proposes a reconfiguration mechanism that prevents the references over the network to the mobile actor from being affected by the move process. The references at the initial and pulling hosts are automatically updated during the move process whereas the references at third-party devices are updated by the initial host via message. In case of disconnection of the initial host upon message reception, the references at the third-party device search for the moved actor via multicasting.

**Structured mobility** The MOM model presents move methods that cleanly align mobility with the language default interaction mechanism of actors (message sending protocol). As result of the execution of a move method, the actor is collocated with the message sender which means that there is no need to explicit refer to any location.

**Secure mobility** The security issues regarding mobility (at the language-level) are taken care of because move methods enforce a two-party contract: an actor can only be moved if it is implementing a move method and if an actor on a remote host is willing to send a corresponding move message. Moreover, involving the actor located at the destination host in the move process mitigates some of the security problems concerning resource rebinding, since it is this actor that decides on which resources to share with the moved actor. Furthermore, message-oriented mobility only supports pull mobility in which the actor is collocated with the sender actor, which prevents actors to move unexpectedly to other devices. Of course, this language-level security



policy can only be enforced when assuming that the language's virtual machine is part of the *trusted computing base*, i.e. we assume that the VM hosting the actor to move and the VM hosting the actor that requested the move can be trusted.

## 5 THE FOLLOW-ME PATTERN

To further illustrate the MOM model, we describe how *follow-me* services can be conceived using this model. We show a mobility pattern where services can be grouped into sessions and where these sessions can be moved towards the user.

As mentioned in the introduction, the main idea of follow-me services is to provide the user with services that “follow” him, hopping from one device to another, adapting themselves to the different contexts found at the places where the user moves. A simple implementation of such a scenario would enable the user to move his services manually between devices, for instance from a fixed workstation to his mobile device (like a PDA). Services can be grouped into sessions, and when a session is moved towards the user, all session services follow.

The MOM model allows the user to logically pull his services from e.g. the workstation to the PDA. We implement a framework for follow-me services using actors. The implementation of the follow-me pattern considers three types of actors: *system* actors which represent the devices and can discover remote sessions and pull them towards the user, *mobile session* actors which group services into a single mobile session and *mobile service* actors which represent any kind of mobile service. Figure 3 shows a sequence diagram of the (purely asynchronous) collaboration between these actors. Dotted arrows represent the movement of an actor to the location of the sender actor. The behaviour of each of the collaborators is described in detail below.

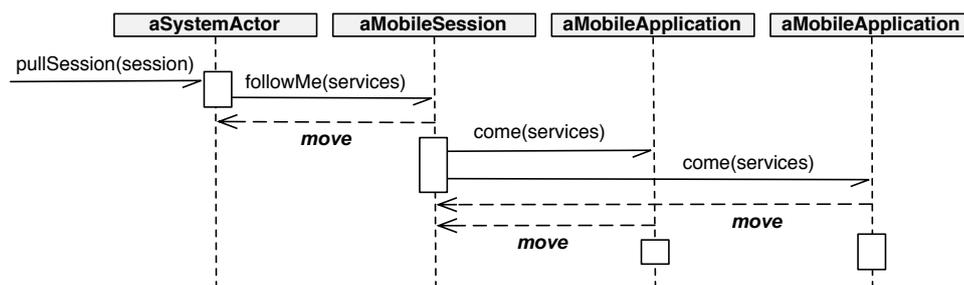
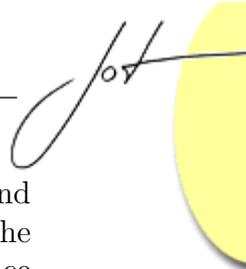


Figure 3: Sequence diagram of the Follow-Me pattern using the MOM model.

### System Actors

A *system* actor is responsible for finding remote mobile sessions and moving them to its device. This actor encapsulates any local services which the user wants to



share with any mobile session pulled towards the device. Mobile sessions are found using a pattern-based discovery mechanism of AmbientTalk which requires that the session actors provide an identification which is broadcast to the network [7]. Once the mobile session with the right identification is discovered, it is pulled towards the system actor when this actor sends it the move message `followMe`. The following code listing shows the behaviour of the system actor:

```
makeSystem(localServices):: actor({
  services: void;
  session: void;
  init():: {
    services:= localServices
  };
  ...
  pullSession(session):: {
    // send the asynchronous "followMe" message to the session
    session#followMe(services)
  }
})
```

The system actor sends an asynchronous `followMe` message to the session (the `#` symbol denotes asynchronous message sending in AmbientTalk). The `services` object is the only reference a mobile session receives when it is asked to move by the system actor. These services are actors that abstract away the direct interaction with hardware and software services (e.g. screen, speakers, databases), similar to *environment* actors in SALSA [24]. For instance, the screen is a service modelled as an actor. A moved actor that wants access to the screen has to be given a reference to the screen actor explicitly in the `services` object it receives upon arrival.

### Mobile Sessions as Actors

A *mobile session* actor is a container for a group of services which have to be moved together. These services are stored by this actor in a simple vector. We assume that the mobile sessions are distinguishable across the network via its user identifier. Part of the implementation of the mobile session actors is shown below:

```
makeMobileSession(userID):: actor({
  services: makeVector();
  init():: {
    broadcast(userID)
  };
  ...
  move followMe(localServices):: {
    // pull the services of this session only if there are
```

```

// no local services providing the same functionality
services.forEach(
  lambda(service)->{
    serviceType: service#getType();1
    localService: localServices.get(serviceType);
    if(void(localService),
      service#come(localServices),
      service:= localService)})
  }
})

```

Our focus is on the `followMe` move method of the mobile session actor, whose body is executed after the session itself is moved by a system actor. During post-move processing, the mobile session actor asks its registered services to come to the new location (by invoking their `come` move method and passing along the services it was given by the follow-me actor) only if there is no service at this location that provides the same functionality. Otherwise, the mobile session rebinds these new services. This example shows how the MOM model can be used to structurally move parts of a service together.

### Mobile Services as Actors

A *mobile service* actor represents any kind of actor that implements a move method named `come`. Upon arrival to the new location the service has the possibility to reconfigure itself by rebinding services passed as parameters in the `come` method. As a concrete example, consider the following *mobile game* actor which represents a (distributed) game application that requires a GUI and speakers to work.

```

makeMobileGame(userID,localServices):: actor({
  guiActor: void;
  speakerActor: void;
  user: void;
  // actor constructor
  init():: {
    bind(localServices);
    user:= userID;
    broadcast(user)
  };
  bind(services):: {
    guiActor:= services.get("gui");
    speakerActor:= services.get("speakers")
  }
})

```

<sup>1</sup>In AmbientTalk, the result of an asynchronous message can be a *future* which corresponds to a proxy for a result that is not yet computed. We refer to [7] for further information about this language abstraction.



```
};  
...  
// a move method  
move come(services):: {  
    bind(services)  
}  
})
```

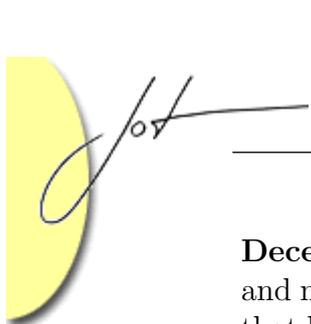
Whenever the actor is moved, it rebinds these services. For the sake of simplicity, we assume that if some service is unavailable the functionality of the actor using such service becomes *disabled* (unbound). For instance, if the speakers cannot be rebound, the game continues without audio.

The scenario of the follow-me services demonstrates how one can concisely define a mobile service using the MOM model. The example features a simple yet representative mechanism to move running services in mobile networks. Without the MOM model, coding these mobile services requires to manually handle concerns like mobility of computational state, reconfiguration of the network, abstractions for structured mobility and security.

## 6 RELATED WORK

In the literature, there are an important number of programming languages and frameworks that have direct support for *object mobility* like Emerald [10] and Obliq [10], *agent mobility* like Telescript [26] and Aglets [12], and *actor mobility* like SALSA [24], Actor Architecture (AA) [9] and ProActive [2]. Although these approaches accomplish some of the four requirements for proactive code mobility in mobile networks identified in Section 2, to the best of our knowledge none of them succeed in all of them. In this section, we discuss the solutions and shortcomings of such mobility approaches and compare them with the MOM model.

**Strong mobility** The mobility of running services have been mainly studied in the context of agents and actors. IBM Aglets [12] and ProActive [2] are Java frameworks that support the mobility of *living* entities (agents and active objects, respectively). Unfortunately, because Java threads are not serialisable, both approaches only supports semi-strong mobility, which as we argued in Section 2 does not suffice for our purposes of proactive service migration. The AA framework [9] provides a form of strong mobility on actors. Similar to the MOM model, this framework preserves the availability of the actor during the move by putting it in *in-transit* state. Messages sent to an in-transit actor are delayed in the original host until the actor notifies its arrival at the new location.



**Decentralised network reconfiguration** In the literature, location updating and message rerouting have been addressed in two different ways: via central servers that keep track of the location of the mobile services, or via *forwarding* proxies. The AA framework, for instance, implements a centralised network reconfiguration mechanism which is supported by a dedicated entity called the *mobility manager*. When actors move to another location, they have to update their location in the mobility manager which takes care of message rerouting. Unlike the network reconfiguration mechanism presented in the MOM model, the reconfiguration mechanism of AA is not an automatic mechanism since it requires that actors explicitly register their location each time they migrate to a new host. SALSA [24] is an actor-based programming language that also uses a naming service in which actors have to register their changes of locations. However, this is a decentralised naming service based on a Chord distributed lookup protocol [22] which is resilient to the network partitions. The ProActive framework proposes a message rerouting mechanism that is entirely based on forwarding proxies. This framework attempts to keep the chain of forwarding proxies as short as possible by means of active collaboration of the sending host where messages that are forwarded are marked so that the receiving objects knows if the sender has the latest location or another one.

**Structured mobility** As previously explained, code mobility has is often achieved by the introduction of `move` instructions and explicit locations. The Telescript [26] programming language, for instance, introduces a move instruction called `go` which requires the notion of *places*. The AA framework has a `migrate` operator which receives as parameter absolute locations such as destination host name or IP address. SALSA provides an `migrate` method defined upon actors which takes as parameter a high-level universal description of the new location. The Emerald [10] object-oriented programming language is a pioneer in exploring the use of relative locations for code mobility. This language provides an `attach` declaration that enables programmers to collocate objects and special parameter-passing styles to allow the argument objects of a remote method invocation to be moved along with the invocation, permanently (*call-by-move*) or temporarily (*call-by-visit*).

**Secure mobility** Obliq mitigates the security issues that stem from `move` instructions by introducing `protected` objects and the notion of self-inflictedness. Put briefly, no object can move a `protected` object except for the `protected` object itself. Obliq provides mobility as the combination of cloning an object on a remote location and aliasing references from the original to the cloned object. Migration to a host requires then the active collaboration of that host as it needs to provide an *execution engine*, a function responsible for creating the clone of the object that wants to move. Telescript tries to mitigate security issues by providing a security model based on capabilities. These capabilities are used for reconfiguration purposes of agents upon arrival to new locations. Such capability-based security model allows places to restrict the actions a visiting agent can perform. Likewise, SALSA



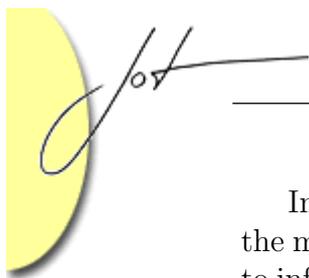
prevents most of the security issues regarding mobility via access control lists [21]. If an actor is not present in the access control list of a remote host, the actor will be rejected upon moving to that host. Similarly, if an actor is not present in the access control list of another actor, it is not allowed to migrate that actor to another host. Unlike the MOM model, this security mechanism explicitly requires programmers to *manually* deal with these access control lists when developing mobile services.

As said before, none of the code mobility mechanisms described in this section fulfil all the requirements for proactive code mobility in mobile networks. However, some of these mechanisms have significantly influenced the MOM model. For example, our model adopts an Emerald-like use of relative locations for code mobility by means of the move methods. In addition, the MOM model provides a security mechanism for rebinding local resources based on capabilities that is reminiscent of Telescript. Finally, we make use of proxy actors for the decentralised network reconfiguration mechanism which is similar to the concept of forwarding proxies of ProActive.

## 7 CONCLUSION AND FUTURE WORK

Within the domain of mobile computing, this paper focuses on the use of proactive code mobility to mitigate the unavailability of services due to network partitions. We identify a number of requirements a code mobility mechanism should accomplish in this context. We require a mobility mechanism that (1) does not hamper the availability of the services, (2) ensures the readability and structure of the code, (3) guarantees the security of the devices involved in the move process, and (4) performs a decentralised network reconfiguration after the move. We subsequently propose an event-driven mobility model, called the *message-oriented mobility* (MOM) model, which exhibits such criteria. The basis of MOM model is the combination of a strong mobility scheme with the actor model. The introduction of an explicit `move` instruction has been circumvented by providing a new kind of method called *move methods*. Furthermore, the model transparently ensures that none of the incoming messages are lost such that services retain their full computational context upon arrival. We illustrate our model by implementing a Follow-me service that allows users to pull services from one device to another.

Although the MOM model can help in tackling some of the challenges for code mobility faced in mobile networks, a number of challenging issues needs to be further explored. For instance, in the MOM model we represent mobile services as single actors. We are currently investigating an extension of the model that enables programmers to also deal with services composed of multiple actors. Moving such services, programmers may want to perform different actions on the actor components rather than just moving them (e.g. rebinding or recreating actors upon arrival). A solution for this case is to introduce an *annotation system* that allows the programmers to declaratively indicate the actions to be executed for every actor of a service when it is requested to move.



In the move process proposed in the MOM model, a proxy is left at the old host of the moved actor to forward local messages (sent from the same proxy's location) and to inform the new location to actors residing in third-party devices. We are currently investigating network reconfiguration techniques of peer-to-peer architectures [19, 17] as more scalable and reliable schemes to avoid chains of forwarding proxies in case of multiple moves of the actor.

The MOM model proposes a two-party contract between the devices involved in a move process as a solution for security at the language-level. This scheme assumes that both devices trust each other. We are currently investigating more advanced security techniques based on capabilities [13, 14] to complement our model. Capabilities allows the mobile service to be sure that only trusted devices can request its move.

## REFERENCES

- [1] G. Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] F. Baude, D. Caromel, F. Huet, and J. Vayssiere. Communicating mobile active objects in java. In *HPCN Europe LNCS 1823*, pages 633–643, 2000.
- [3] S. Berger, H. Shulzrine, S. Sidiroglou, and X. Wu. Ubiquitous computing using sip. In *NOSSDAV 03: Proceedings of the 13th international workshop on Network and operating systems support for digital audio and video*, New York, NY, USA, 2003.
- [4] L. Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures: First International Conference, FOSSACS '98*, 1998.
- [5] J. Dedecker. *Ambient-Oriented Programming*. PhD thesis, Vrije Universiteit Brussel, 2006.
- [6] J. Dedecker and W. Van Belle. Actors for Mobile Ad-hoc Networks. In *International Conference on Embedded and Ubiquitous Computing EUC2004*, 2004.
- [7] J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, and W. De Meuter. Ambient-Oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, 2006.
- [8] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
- [9] M.-W. Jang. The Actor Architecture. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
- [10] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
- [11] J. A. Landay and G. Borriello. Design patterns for ubiquitous computing. *IEEE computer ubicomp*, August 2003.
- [12] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998. <http://aglets.sourceforge.net>.
- [13] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.
- [14] M. Miller, C. Morningstar, and B. Frantz. Capability-based financial instruments. In *Proceedings of Financial Cryptography*. springer-Verlag, 2000.



- [15] D. S. Milojevic, F. Douglis, and R. G. Wheeler, editors. *Mobility: Processes, Computers, and Agents*. ISBN: 0-201379-28-7. Addison Wesley and ACM Press, 1999.
- [16] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *Mobile Computing and Networking*, pages 32–43, 2000.
- [17] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.
- [18] I. Satoh. Physical mobility and logical mobility in ubiquitous computing environments. In *MA '02: Proceedings of the 6th International Conference on Mobile Agents*, pages 186–202, London, UK, 2002. Springer-Verlag.
- [19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.
- [20] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.
- [21] R. Toll and C. Varela. Mobility and security in worldwide computing. In *Proceedings of the 9th ECOOP Workshop on Mobile Object Systems*, Darmstadt, Germany, 2003.
- [22] C. Tolman and C. Varela. A fault-tolerant home-based naming service for mobile agents. In *XXXI Conferencia Latinoamericana de Informática*, Cali, Colombia, 2005.
- [23] T. Van Cutsem, J. Dedecker, S. Mostinckx, E. Gonzalez, T. D'Hondt, and W. De Meuter. Ambient references: Addressing objects in mobile networks. In *Proceedings of the Dynamic Language Symposium - OOPSLA '06*, Portland, U.S.A., 2006.
- [24] C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.
- [25] G. Vigna. Mobile agents: Ten reasons for failure. In *IEEE International Conference on Mobile Data Management (MDM '04)*, pages 298–299, January 2004.
- [26] J. E. White. Telescript technology: Mobile agents. *Software Agents*, 1996.

## ABOUT THE AUTHORS



**Jorge Vallejos** is a PhD student at the Programming Technology Laboratory of the Vrije Universiteit Brussel, Belgium. His research interests are language design and implementation for mobile and context-aware computing. He is funded by the Flemish project of Context-Driven Adaptation of Mobile Services (CoDAMoS). His coordinates can be found at <http://prog.vub.ac.be/~jvallejo>.



**Tom Van Cutsem** is a PhD student at the Programming Technology Laboratory of the Vrije Universiteit Brussel, Belgium. His research interests are language design and implementation, distributed programming and reflective architectures. He is funded by a doctoral scholarship of the Fund for Scientific Research, Flanders (F.W.O.). His coordinates can be found at <http://prog.vub.ac.be/~tvcutsem>.



**Elisa Gonzalez Boix** is a PhD student at the Programming Technology Laboratory of the Vrije Universiteit Brussel, Belgium. Her research interests are memory management for distributed systems. More specifically, her research is focused on how to reconcile distributed garbage collection with the characteristics of mobile ad hoc networks. She can be reached at <http://prog.vub.ac.be/~egonzale>.



**Stijn Mostinckx** is a PhD student at the Programming Technology Laboratory of the Vrije Universiteit Brussel, Belgium. In the past two years, he has been involved in the development of the ambient-oriented programming paradigm and AmbientTalk. His main research interest is the use of exception handling techniques to deal with partial failures. His coordinates can be found at <http://prog.vub.ac.be/~smostinc>.



**Jessie Dedecker** Jessie Dedecker is a post-doctoral research assistant at the Vrije Universiteit Brussel. His research interests are programming language design and implementations, distributed computation and biologically-inspired programming models. He is funded by the Interuniversity Attraction Poles Programme, Belgium. Further information can be found at <http://www.dedecker.net/jessie>.



**Wolfgang De Meuter** is a professor at the Vrije Universiteit Brussel. He has been active in the field of object-orientation since the early nineties. His research interests include programming languages and their evaluators, aspect-oriented programming, meta-programming and more recently also language constructs for ambient-oriented systems. He has organized numerous successful workshops at previous ECOOP's and OOPSLA's. His coordinates can be found at <http://prog.vub.ac.be/~wdmeuter/WolfHome>.