An API for high-level software engineering of distributed and mobile applications

Jean-Paul Arcangeli, Christine Maurel, Frédéric Migeon

Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier
118, route de Narbonne
31062 Toulouse Cedex 4
France
email: Firstname.Lastname@irit.fr

Abstract

This paper proposes actors as a model for mobile agents. Because of their intrinsic autonomy (thread integration) and behavior changing ability, added to asynchronous communication, actors are naturally mobile entities, and moving actors has no effect on their initial semantics. Then, we propose a standard API on top of Java and RMI, called JavAct, for distributed and mobile applications programming. JavAct is actor-based and allows nearly immediate and strong agent mobility. Mobile actors are localized by means of a forwarding chain technique. Network-level references of actors contribute to location independence of computations. Additionally, in this paper, we give an overview of some related programming systems.

1. Introduction

From clusters of workstations to wide-area networks, wired or wireless, new physical supports for distributed applications become available. Today, new applications are emerging in several domains (e-commerce, data mining, networks management, distributed databases, ...) demanding well-adapted programming paradigms and abstractions. Among them, mobile agents [10] are autonomous software entities which can decide to move and relocate themselves in the network, carrying both their code and execution state. They perform tasks on behalf of a user, mobile or not. Ideally, any application using mobile agents could be programmed without them. The main interest in the use of mobile agents is to replace remote interactions with servers by local ones, in order to reduce communication costs. We think also that using mobile agents can increase expressiveness in distributed programming.

Mobile agents have inherited computational and decisional autonomy from intelligent agents. They are able to interact with other agents and possibly with their environment. Compared to mobile code, mobile agents allow to move computations. Compared to process migration which is decided by a runtime manager, mobile agents decide to move by their own : such a mobility is called pro-active. Agent mobility can also be reactive, but the use of mobile agents is not restricted to load balancing or fault tolerance achievement.

Software engineering with mobile agents is not so easy. The need of flexibility and adaptativity to the context (heterogeneity, network and hosts resources dynamic variations) increases significantly complexity and costs of software development and maintenance. Thus, expressive high-level programming models and APIs for mobile agents programming must be developed. In this paper, we propose actors as a model for mobile agents, and JavAct as an API on top of Java for programming distributed and mobile applications.

The paper is organized as follows. Section 2 discusses actors as a mobile agent model, in comparison to objects. Section 3 presents JavAct, a Java API that we have developed, for distributed and mobile actor programming. Section 4 describes the main features of some platforms for actor or mobile agent programming. At last, section 5 mentions additional works and prospects.

2. Modeling mobile agents with actors

2.1. Facilities for mobile agents programming

Languages and platforms for mobile agents must provide mechanisms and abstractions for :

- Concurrency and synchronization.
- Agent migration (with code, data and state) in an heterogeneous context.
- Network-level identification and localization.
- Point to point asynchronous communication.
- Security (of both agents and hosts).

According to [10], mobility is strong or weak depending on the ability of capturing and restoring the agent's execution state when moving. For example, mobility is weak in Java because it is impossible to access to thread stack values and to serialize threads. The problem of mobility degree is mainly a problem of expressiveness.

2.2. Programming mobile agents with objects

Objects are good candidates for the implementation of agents, and existing mobile agents platforms are, in many cases, based on concurrent objects enhanced with mobility mechanisms (see section 4). But introducing mobility in the object model is not transparent and has effects on mechanisms like synchronization and method invocation. Due to the orthogonality between activities (threads) and objects, mobile objects invocations have to be processed one per one, to avoid several threads to manipulate a single object ; otherwise, all the threads have to be completed before moving the object. Because of network latency, synchronous object invocation mechanism must be turned into asynchronous message passing.

Additionally, processing messages serially can lead to a periodic easy-to-handle state which is convenient for strong mobility. But, consequently, some latency is introduced between the processing of the mobility primitive and the actual object movement that is delayed in order to reach the convenient state (see sections 4.1 and 4.3).

Hence, adding mobility to standard objects may unfortunately lead to some semantic changes in applications and to weak reusability of codes.

2.3. Mobile actors vs. mobile objects

Here, we argue for actors, rather than standard objects, for mobile agent programming. The use of actors [1] for concurrent and distributed programming is not new. Actors are autonomous anthropomorphic entities which react to messages and process them one per one. When processing a message, an actor can create other actors (dynamically), communicate by asynchronous point-to-point message passing with other actors that it knows, and change its own behavior (define the behavior for the processing of the next message). Behavior changing may be useful for roaming web agents because it provides a way for evolution and learning.

Thus, actors can be seen as active objects with the ability of changing their interface. In applications, actors as agents can be both clients and servers. Thanks to autonomy, asynchronous message passing and behavior changing, they are naturally mobile units :

- Autonomy is an important property that agents must have for mobility self-decision. The encapsulation of data and methods in the actor's private behavior (a closure) conceptually guarantees privacy and integrity. Actors encapsulate not only programs and data, but also activity : there is a single activity per actor (or possibly per behavior in a finer grained way). Actor systems are multi-threaded, but synchronization problems are hidden from the programmer.
- Asynchronous message passing is another important feature for mobile agents, because synchronous communications are expensive and hard to maintain in the context of wide-area or wireless networks (standard call/return is unadapted because of latency and failures).
- Actor's behavior includes all its data and code. At behavior changing time, actor's state is fully contained in the behavior (and in the mailbox), and so, easily capturable and transferable. At this transitional moment, there is nothing more in the execution state. Movement is so delayed (only) to the end of the current message processing. Actor's mobility is based on a remote creation of an actor from the behavior intended to process the next message. Consequently, the actor moves carrying both acquired knowledge and experience. Thus, actors move but behaviors, during their execution, don't move.

Localization of moving agents is possible using either a name server or a forwarding system. In actor systems, each actor has a unique reference (a postal address) and localization is natural by means of a chain of forwarders. Every time the agent moves, the local actor remains after the remote creation and becomes a proxy for the agent : it receives messages for the agent and forwards them to the remote reference. Such a method allows also messages, stored in the mailbox before moving, to reach the agent after it has moved. The mobile actor reference remains valid even if moving is in progress or if the actor is remote. So, mobility is transparent for communications (code of sender agents hasn't to be modified whether the target agent is mobile or not). However, this basic protocol is known to be few efficient and fault sensitive, because of the multiple relays : several kinds of optimization can be provided.

In conclusion, we can assert that enhancing actors with mobility does not involve semantics changes.

3. JavAct : an API for distributed and mobile actors in Java

Thanks to its object-orientation and since it offers tools and abstractions to deal with heterogeneity, concurrency, security and remote interactions, Java supports most of the existing platforms in the domain.

We have developed JavAct, a standard Java library for actor-based programming of concurrent, distributed and mobile applications. JavAct is a successor of a platform for distributed actor programming which had been developed in our group at the end of the 80's [13], based on a functional language and on a distributed virtual machine. JavAct has been designed to be easy-to-use, minimal in code and so maintainable at low cost : consequently, we rejected the use of pre-processing or post-processing techniques, and any JVM modification. Current version runs under SDK 1.3, and JavAct programmers can use any standard Java toolkit environment. JavAct classes rely on Java libraries : RMI, threads (*java.lang.Thread*), object serialization (*java.io.serializable*) and reflexion (*java.lang.reflect*). The scheduling of threads is left in charge of the JVM.

A set of places (which can dynamically change) constitutes a domain on which applications run. A place -i.e. a virtual machine- is either a physical or a logical site ; it can be seen as an actor server providing environment and resources for actor execution. If needed, it is possible to simulate distribution by creating several places on a single physical site. It is not necessary to change the code to distribute an application ; the same program can be used in a local environment or in a distributed one.

JavAct is portable, and today, it mainly runs on a network of Sun workstations with Solaris. It can be used by junior programmer in Java who knows a little of actors. With JavAct, programmers write high-level code and are not interested in low-level mechanisms such as threads, synchronization, RMI or Corba, ... Mixing object-style and actor-style of programming is possible in order to benefit from both advantages.

3.1. JavAct programming abstractions

Programming actors with JavAct mainly consists in defining behaviors. This is done by extending the *Behavior* class (for the programmer, it is the main class of the JavAct library) with the specific methods of the programmed actor : JavAct behaviors are sequential objects and each public method is a possible response to a message.

The *Behavior* class defines methods for actor creation and distribution, communication, behavior changing and mobility :

- *RefActor createOn(Behavior b, Place p)* allows actor creation on the place *p* from the behavior *b* and returns the actor's reference ;
- *RefActor create(Behavior b)* abstracts location in actor creation ;
- *void send(Message m, String methodName, RefActor act)* allows to send asynchronously a message *m*, intended to be processed with the behavior's method *methodName*, to *act* ;
- *void become(Behavior b)* defines the next behavior of the actor (if not used, actor's behavior remains unchanged) ;
- *void becomeOn(Behavior b, Place p)* allows pro-active mobility as described before.

Actor are "network" objects which can be referenced from anywhere in the network of places. Messages can be sent locally or remotely in a location-transparent way. Let's notice that message passing of behaviors is possible. When behaviors are arguments of remote interactions (remote creation, behavior sending, mobility), they are passed by value, but inner pointed actors are passed by reference : actors cannot be moved by value. Thus, no special care is necessary when programming distributed and mobile actors in JavAct (unlike Java RMI objects).

JavAct programs are sets of classes defining behaviors and messages (by implementing the *Message* interface) ; a main program is used to launch the application.

3.2. Implementation

This paragraph gives details on JavAct implementation which can be useful for understanding but which are not compulsory for programming.

- **Distribution and communication**. Distribution features rely on Java Remote Method Invocation (RMI) and serialization. Installing JavAct network consists in launching on every place both a name server (RMI registry) and a virtual machine running a RMI remote object dedicated to actor creation. This object, registered in the name server, offers a public method, which can be remotely invoked, for actor creation : when it is requested with a behavior, it performs local actor creation and returns the reference to the caller (note that, for safety and efficiency reasons, behavior creation on the caller site and then serialization could be avoided and replaced by a direct remote creation).

Actors are RMI remote objects too, which public remote interface is restricted to a method for message reception. So, an actor reference is a network reference to which messages can be sent directly, remotely or not (as actors are only known by their reference, it is unnecessary to register actors in name servers). Receiving a message consists only in storing it in the actor's mailbox.

- **Message processing**. Inside actors, mailboxes and behaviors are *private* Java objects handled by threads through a procedure for executing actor life-cycle. After a message is got from the synchronized mail queue, reflexion is used to build the behavior's method call, from the sent message and *String* method name (as message passing is asynchronous, methods are typed *void*, and so, there is no result to manage). However, Java compiler can't be used to check conformity between the sent message and the behavior's methods. Several politics are possible for messages which cannot be processed : exception handling, message giving up, message storage in a second queue.

- **Threads**. In JavAct current version, there is one thread per actor. Practically, the number of actors is so limited (and consequently the size of the application) depending on Java runtime. As threads and actors are loosely coupled in the current implementation, we think we will be able to modify JavAct execution model at low cost. In order to avoid the limitation in number of threads, we plan to use a fixed-size pool of threads executing an independent pool of actors. Thus, applications could scale up, and furthermore, thread-pooling could improve local efficiency as in Voyager [14].

- **Efficiency**. In this paper, we few discuss JavAct efficiency : the most important factor for efficiency of mobile applications seems to lie mainly in the amount of data moved over the network, more than in intrinsic performance of the platform itself. In fact, we emphasize on expressiveness and software engineering costs rather than on execution costs. Nevertheless, concerning the platform, we think that localization mechanism, security policy and thread management techniques are essential.

3.3. Mobility degree

In a way, JavAct mobility mechanism (tied with behavior changing) provides strong mobility in Java, since mobile actors resume remotely at the execution point where they stopped.

However, mobility is quite immediate or not depending on the grain of the behavior method. As a behavior method is not suspendable until it finishes, the grain could practically be too coarse. But this grain may be user-defined. It is possible to split a method in two pieces and put them in two behaviors that will be run by the actor sequentially, the first one before migration, the second one after. But, programming that can be a little tricky, and defining the behavior for remote execution can be equivalent to do explicit checkpointing.

In fact, mobility degree does not depend on the actor model but on the language for behavior programming (Java in our case). If it was a functional language with the ability to reify and manipulate continuations, the problem would be quite different.

3.4. Semantics and security

A moved behavior contains references to actors which, in our system, do not have to be transformed. Like Obliq [7], JavAct relies on a mechanism of network-wide lexical scoping : the main advantage is the preservation of the semantics of moved actors and the independence between computation and locality. This allows to reason upon the programs independently from the location of activities. Moving is also more efficient because referenced actors do not have to be serialized. Conversely, it is known that this solution can also lead to a significant increase in network traffic which could go against the benefit awaited from mobility.

Until now, we few considered security features. In fact, JavAct security model is twofold. It relies on Java sandbox security model and on the network-wide lexical scoping property : a mobile actor can't access to a resource of a host if the reference has not been communicated to the actor explicitly. In addition, privacy of actor components is at the basis of its integrity.

3.5. Experiments

Experimenting mobile actors and JavAct for the engineering of several usual mobile applications is underway. Some test application, and among them an e-commerce one, have been developed and experimented on a cluster of workstations. Considering our present experience, it seems that no special care is needed for the programming of mobile agents with actors and JavAct.

4. Related programming systems

In this section, we describe briefly some of the existing platforms for either actor or mobile agent programming, which are related to our work.

4.1. Actor Foundry and Salsa

In the last few years, G. Agha's research group at OSL, developed two platforms for distributed programming : Actor Foundry [15], and then Salsa [2].

Although both platforms are actor-based, behavior changing ability has been removed : in our opinion, the result is a lack of expressiveness even if Java allows to change object state dynamically and consequently to program history sensitive actors. Thus, actor mobility is not linked to behavior changing like it is in JavAct. On the other hand, proving properties of programs is much more difficult when behavior changing is used.

- Actor Foundry is a Java API for distributed computing. Programming actors with Foundry consists in extending the Actor class. Public methods serves as target for messages, like in JavAct. Actor Foundry runtime system is a set of specific interacting modules (scheduler, request handler, name service, message transport layer, ...).

- Salsa is not a Java API but a dialect on top of it, dedicated to world-wide computing. A pre-processing step is needed to translate Salsa programs in Java. Universal actor names (UANs), which are world-wide global identifiers linked to actual locations of actors, provide transparent agent localization and migration. Communications rely on a specific Remote Message Sending Protocol (RMSP) built on top of TCP/IP.

  Salsa actors move on places (theaters in Salsa terminology) in response to an asynchronous bufferized migrate message. Even if actor's state is saved and restored remotely, the message sending mechanism introduces additional latency compared to JavAct. When an actor migrates, actor acquaintances are passed by reference while Java serializable objects and primitive types are passed by value ; environment actors (like standardOutput) are automatically bound, after migration, to the resources of the new place.

In addition, in [2], authors propose capsules for actor coordination called Cyborgs (cybernetics organisms) as a basis for the construction of distributed intelligent multi-agent systems.

4.2. JMAS

JMAS [6] is an actor-based Java API for global computing on networks. JMAS runtime relies on a virtual network of distributed runtime managers (with specialized schedulers, load balancers and class loaders). Computations are performed within actors which can move from one computer to another. Here, the goal is to distribute the load over the network and exploit the large amount of computing power available ; mobile computing is based on message-driven actor migration. However, even if JMAS uses mobile code and actors, it is not a tool for mobile agent programming.

4.3. ProActive

ProActive [8][4] is a Java library for parallel, distributed, and concurrent computing. It is based on RMI Java standard library and is made of standard Java classes which don't require any change to the Java Virtual Machine, preprocessing or compiler modification.

ProActive extends Java with active mobile objects and asynchronous message passing based on automatic future synchronization. This wait-by-necessity communication is implemented with future objects which store responses. Remote calls are stored and synchronized in a queue. Like in Salsa, migration requests are mixed with these calls. This insures that when the object will process it, the execution context is well defined (no other thread is processing the object) but introduces latency. However, it is a smart way for introducing object mobility but it suffers a change of semantics when an object is turned remote and active.

4.4. Aglets

Aglets [11] is a Java API developed by IBM Corporation for the development of Internet applications based on mobile agents. It relies on a specific protocol for aglet transfer (ATP), and on specialized class loader and security model.

Aglets are mobile agents executed within a place (a context in Aglets terminology), which are able to move reactively or pro-actively. Mobility is weak and methods to be executed *on dispatching*, *on arrival* and *on reverting* must be programmed. An aglet can communicate with one another by synchronous or asynchronous with future message passing ; multiple aglets coordination is also possible thanks to multi-casting. Aglets are identified globally and proxies are used to hide aglets real location and provide transparency. In addition, some design patterns like the itinerary pattern are provided.

4.5. Voyager

ObjectSpace Voyager [14] is not just a mobile agent system but a Java-based platform developed in order to make distributed applications easier to design, develop and deploy. It provides an ORB compatible with CORBA, RMI and DCOM and so emphasizes on interoperability.

Voyager enables object mobility (all threads running the object have to be completed before the object moves) and supports autonomous mobile (reactive or pro-active) agents. Several communication protocols between agents are available. Localization is based on an optimized forwarding protocol. As in Aglets, mobility is weak and methods for moving and restoring state must be programmed. At last, Voyager relies on a thread-pooling technique to avoid repetition of thread creation and destruction.

4.6. Obliq

Obliq [7], developed at Digital SRC is an object-oriented concurrent dynamically typed interpreted language which design is prior to the Java boom. It is based on distributed lexical scoping, shallow remote copy and aliasing. Distribution must be expressed but computations are network transparent. Any object is a network object which can be accessed directly or by means of a name server. As we said before for JavAct, network-wide scoping improves the control on application complexity and safety, but can also affect performance.

5. Works around and future works

High-level expressiveness of actors has led us to explore methods and tools for the development of concurrent application by separating "logical" applicative programs from resource allocation ones.

With our previous actor platform, we studied reflection and meta-level architectures [12]. Reflection is the ability for a system to reason and act upon itself dynamically. We obtained it through a self-representation materializing both some of the structural and computational features of actors (or groups of actors) and tools to manage them. Such a technique is called Meta-Object Protocol (MOP). It leads to a natural separation of concerns in application design and to two levels of programming :

- a "base-level" which is used for the application description,
- a "meta-level" where execution features are programmed ; meta-level components constitute a programmable open layer between the application and the runtime system.

We use reflection and MOPs for modular and expressive programming of actor scheduling [5]. As mobility protocols (security, localization, ...) depends on the characteristics of applications, and as mobile applications need to adapt to software and/or hardware heterogeneity, and to dynamic variations of runtime conditions, a dedicated level for separate customization and adaptation is helpful. So, we think that reflection is a useful software engineering tool for the introduction of flexibility in mobile applications, while limiting complexity and maintenance costs, and increasing portability and reusability. In [3], we presented some examples of the use of reflection for the customization of security policy and localization optimization of mobile actors. Presently, we are working on the integration and the adaptation of these techniques to JavAct.

Furthermore, tools for safety errors detection, based on static analysis, are currently being studied and developed in our group [9].

References

[1] G. Agha. *Actors : a model of concurrent computation in distributed systems*. M.I.T. Press, Cambridge, Ma., 1986.

[2] G. Agha, N. Jamali, and C. Varela. Agent naming and coordination : Actor-based models and infrastructures. In A. Ominici, F. Zambonelli, M. Klusch, and R. Tolksdorf, editors, *Coordination of internet agents*. Springer-Verlag, 2001. to appear.

[3] J.-P. Arcangeli, L. Bray, A. Marcoux, C. Maurel, and F. Migeon. Reflective actors for mobile agents programming. In *ECOOP'2000 Workshop on Reflection and Meta-level Architecture , Cannes, France*, 2000.

[4] F. Baude, D. Caromel, F. Huet, and J. Vayssière. Communicating mobile active objects in java. In *Java in HPC worshop, HPCN'00*, number 1823 in Lecture Notes in Computer Science, pages 633–643. Springer-Verlag, 2000.

[5] L. Bray, J.-P. Arcangeli, and P. Sallé. Programming concurrent objects scheduling strategies using reflection. In *ACM ICS Workshop on Scheduling Algorithms for Parallel and Distributed Computing -From Theory to Practice- , Rhodes, Greece*, pages 7–11. ACM, 1999.

[6] L. Burge and K. George. JMAS : a Java-based mobile actor system for distributed parallel computation. In *Proceeding of the 5th Usenix Conf. on Object-Oriented Technologies and Systems, San Diego, Ca.*, 1999.

[7] L. Cardelli. Obliq : a language with a distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[8] D. Caromel, W. Klauwer, and J. Vayssière. Towards seamless computing and meta-computing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, J. Wiley and sons, 1998.

[9] F. Dagnat, M. Pantel, M. Colin, and P. Sallé. Typing concurrent objects and actors. *L'Objet*, 6(1):83–106, Hermes, Paris, 2000.

[10] A. Fuggetta, G. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[11] D. Lange and M. Oschima. *Programming and deploying Java mobile agents with aglets*. Addison-Wesley, 1998.

[12] A. Marcoux, C. Maurel, F. Migeon, and P. Sallé. Generic operational decomposition for concurrent systems : semantics and reflection. *Parallel and Distributed Computing Practices*, 1(4):49–64, Nova Science Publishers inc., 1998.

[13] A. Marcoux, C. Maurel, and P. Sallé. AL1 : a language for distributed applications. In *Proceedings of the workshop FT-DCS'88, Hong Kong*, pages 270–276. IEEE-CS Press, 1988.

[14] ObjectSpace Inc. *Voyager 4.0 Documentation, http://support.objectspace.com/doc/index.html*, 2000.

[15] Open Systems Laboratory (University of Illinois, Urbana-Champaign). *The Actor Foundry, http://osl.cs.uiuc.edu/foundry/index.html*, 1999.