# Towards a Distributed Actor Development Framework

## Optimized Communication using Mobile Actor

NajatRafalia,
IbnTofail University, Science Faculty, Kenitra, BP. 133,
Morocco, Computer Science Department
Laboratory of Computer Sciences Research and
Telecommunication, arafalia@yahoo.com,

Jaafarbouchabaka
IbnTofail University, Science Faculty, Kenitra, BP. 133,
Morocco, Computer Science Department
Laboratory of Computer Science Research and
Telecommunication,abouchabaka3@yahoo.fr

***Abstract*----In this paper, we present a mobile implementation of a strategy of communication between actors in a distributed environment. This strategy is used bythe $\lambda_{AD}$-calculus witch is an elementary functional distributed actor language [1]. It's based on a static analysis which allows determining the parts of a message that must be transmitted. The actors we consider have a functional script and manipulate the terms of the $\lambda_{AD}$-calculus. The expressions of this language correspond to those of the λ-calculus extent by some actor primitives.The mobile implementation of communication strategy witch we present in this paper is based on mobile actors. In the distributed actor system, each host has its mobile actor witch can migrate to the other hosts with the parts of messages, that must be transmitted outside. Mobile agent migrates to the hosts which have a lot of interactions with his host. In the case of the low interactions, this approach become less efficient, it's penalized by the size of the mobile actor. In this second case, the messages towards distant sites are addressed to the server actor of the sender site in order to treat and send them to their destination. In the receiver site, the server actor treats the external messages that arrive and transmit them to their receivers.**

***Key terms: Actor Model, Communication, Distributed System, Concurrent Programming, Functional Programming, mobile actor.***

## I. INTRODUCTION

We consider a distributed system of actors. The actors we consider have a functional script and manipulate the terms of the ***$\lambda_{AD}$-calculus*** which is an elementary functional distributed actor language [1], described in section3. The expressions of this language correspond to those of the λ-calculus extent by some actor primitives.

Actors communicate by exchanging messages. A message is a functional term. The message transmission causes the message address to be put in the receiver mail queue. This implantation doesn't involve any problem when the actors are in the same site because they share the common memory. If the actors are in some distant sites, they could not get to each other site memory. So, sending the message reference is insufficient.

We have integrated in the ***$\lambda_{AD}$-calculus***a lazy strategy of message communication between actors [1]. For each actor susceptible of receiving a message m which is a functional term, we determine the part of m, that must be sent. This is accomplished by a static analysis of the application code.

In this paper, we present a mobile implementation of this communication strategy, based on mobile actors. In the distributed actor system, each host has its mobile actor witch can migrate to the other hosts with the parts of messages, that must be transmitted outside.

Mobile actor is a soft entity that can migrate during the execution from host to other host. It migrates with its code, its execution state and its data. The mobility is controlled by the application not by the execution system. Le but of the migration is to reach distant data and resources, to do the treatment locally and to move just necessary data.

So, mobile agent migrates to the hosts which have a lot of interactions with his host. In the case of the low interactions, this approach become less efficient, it's penalized by the size of the mobile actor. In this second case, the messages towards distant sites are addressed to the server actor of the sender site in order to treat and send them to their destination. In the receiver site, the server actor treats the external messages that arrive and transmit them to their receivers.

Using mobile actor involves the optimization of the traffic because exchanged messages become locals and discharge the network. Mobil actors can use the resources of distant hosts, so, it discharges its initial host. The size of the mobile actor adds a surcharge when it's migrating but this charge is compensated by local interactions.

## II. THE ACTOR MODEL

The actor model is a model of concurrent computation. It was first described by the group of Massachusetts Institute of Technology (MIT)[9].

Actors are independent concurrent objects that cooperate and interact by sending asynchronous messages.

An actor is completely described by :

- Its **mail address**, to which there correspond a sufficiently large mail queue, and
- Its **behavior**, which is a function of the accepted messages

When an actor machine receive a message in a mail queue, it will create a new actor machine, $\mathbf{X_{n+1}}$ **(become(**$\mathbf{X_{n+1}}$**))**. This new machine will carry out the replacement of the actor. The replacement behavior will process the $(n+1)^{th}$communication. The mail address of the actor remains invariant.
The actor may also send a communication (**m**) to a specific target actor (**a**) (**send(a,m))**. It also creates a new actor with an initial behavior $\mathbf{X_0}$**(create(**$\mathbf{X_0}$**)).**

## III. $\lambda_{AD}$-CALCULUS: AN ELEMENTARY FUNCTIONAL DISTRIBUTEDACTOR LANGUAGE

The ***$\lambda_{AD}$-calculus*** is a distributed extension of the $\lambda_{AD}$-calculus, for the actor model, described in [1]. The λ-calculusis an elementary functional language [4].The $\lambda_{AD}$-calculusconstitutes a low level functional distributed actor language to which the high level actor languages could be compiled. The new version of the $\lambda_{AD}$-calculus integrates the instruction **migrate(Hi)**allowing the migration of an actor from host to another site.

To represent data and the messages, $\lambda_{AD}$-calculusintegrates a structure of term which corresponds to a tree. It also integrates a pattern-matching mechanism in order to recognize the messages.
The actor behavior is an expression of the $\lambda_{AD}$-calculusextended by the primitives for the creation and the manipulation of the actors and those for the pattern matching and the construction of the terms.

### *A. $\lambda_{AD}$-calculus Syntax*

The $\lambda_{AD}$-calculusexpressions are constructed by the terms of the algebra engendered by a set of constructors, a set of variables and the mechanisms of abstraction and application.
The abstraction on a variable of the $\lambda_{AD}$-calculus is generalized to an abstraction on a pattern which is a term of the algebra engendered by the constructors. The $\lambda_{AD}$-calculus contains also the primitives **send**, **create, become**and **migrate**for the manipulation of the actors.

- **send(a,m)** to send the message **m** to the actor **a.**
- **become(b)** to replace the behavior of the actor executing this primitive, by the behavior **b.**
- **create(b)** to create a new actor with an initial behavior **b**.
- **migrate(Hi)** to migrate to a host number i

In the expressions of the $\lambda_{AD}$-calculus, it's necessary to distinguish the actor behaviors which could contain functional computations, the primitives **send** and **become**, from the messages or communicable values which are the results of a pure functional computation. These values could be described by the following syntax:

m = x variables, symbols, address, numbers
| C(m,m,….,m) construction of terms
|Create(B{Acquaintances}) creation of an actor

| self the individual actor address

Note that the messages must be partially valued in order to be filtered. The mechanism of patter-matching takes charge of this lazy valuation.
The result of the **create** operation is an actor address which is a communicable value.
The **self** variable designates implicitly the actor which executes the behavior and allows to this actor to send to himself a message.

The behaviors have the following syntax:

B{acquaintances}= λ**P.Fa**abstraction on a pattern
|λ**P.Fa,** λ**P.Fa, ….,**λ**P.Fa**composition of abstractions on several patterns

The actions **Fa**have the following syntax:

**Fa** = send(**m**,**m**); **Fa**
| create(B{f'{acquaintances}) ; **Fa**
| become(B{acquaintances})
| migrate(Hi)

**Example 1**: **consultation and change of the cell value**

The following behavior **Cell{v}** is a behavior of an actor **Cell** which sends the initial value "**v**" to an actor **a** when it receives the message **Pair(Get,a)**. When **Cell** receives the message **Pair(Set,n)**, it changes its initial value "**v**" by the value "**n**"**.**

**Cell{v} =** λ**Pair(Get, a). send(a,v);become(Cell{v})**
λ**Pair(Set, n). become(Cell{n})**

The following expression creates an actor 'A' and sends him the messagepair(Set,4) :
A = create(Cellf{0}), send (A, Pair(Set,4))

### *B. Lazy strategy of message communication*

The execution of a transmission **send(a,m)**, consists of the valuation of the receiver actor **a** and that of the message **m**.
If the valuation of the actor **a,** detects that this later is in a distant site, then, the transmission of the message address is not sufficient because distant actors could not get to each other site memory. In this case we opt for a lazy transmission between distant sites.
The general problem is presented as follow:

*Given a message C($C_1$,;$C_2$;..., $C_n$) destined to an actor* **a**, *what are in this message the necessary levels to accomplish the pattern-matching and the treatment of the message by the actor* **a**.
Our lazy communication strategy consists of two phases:

- **Static analysis phase**: it's accomplished at the time of the compilation. It allows determining the parts of the message, that are necessaries for the pattern-matching and the treatment of this message. These parts are expressed in the level number of the tree which represents the message.
- **Dynamic transmission phase**: because the static analysis is not always informative, several parts of the message are not detected necessary at the time of the compilation. So, this phase allows completing these needs in the execution.

*Static Analysis*

The static analysis concerns in fact, all the patterns of the application behaviors. The analysis of an initial behavior involves, through the **become**primitive, to analysis the replacement behaviors starting by this initial behavior. It consists of four principal steps:

- **Marking**: through each pattern, the marking phase marks the necessary parts in a message which will be filtered by this pattern and treated by the action corresponding to the successful pattern-matching. An action corresponds to a sequence of several **send** and several **create**, ended by a **become**.

- **Flattening :**marking is don behavior by behavior. A part can be necessary in a behavior and not necessary in other one. The flattening step allows to "**flatten"** the results of marking concerning the same pattern which appears in an initial behavior and in other replacement behaviors from this initial behavior. The ended set of replacement behaviors can be determined through the application code. This warrants the termination of our algorithms.

- **Compilation of the patterns**: the necessary in a pattern is expressed by a number of levels. This phase consists of associating to each pattern the number of its necessary levels.

- **Compilation of the send**: the ad-equation between the patterns and the messages **m** figuring in the **send(a,m)**, and the use of the precedent phase results, allow to compile the transmissions **send(a,m)** into **send(a,m,L)**, where **L** is the number of **m** levels which are necessaries to accomplish the pattern-matching and the treatment of **m** by **a**.

The detail of the static analysis phases is presented in [1 ].

The static analysis determines for each class of actors having the same initial behavior $\mathbf{b_i}$, the number of necessary levels in a message **m** which can be filtered and treated by the behavior $\mathbf{b_i}$.
We group the different values of **NecLev**in a table called **table of needs** (see table 1). In this table, each value **L=NecLev($b_i$i,[m])** corresponds to the number of levels in the message **m**, which, each actor having the initial behavior **b**i, needs in order to value **($b_i$ m)**.
**m**is the message filtered by the pattern **[m]**.
If the message **m** is not filtered neither by bi nor by any replacement behavior obtained from $b_i$ then **NecLev($b_i$,[m])**=0.

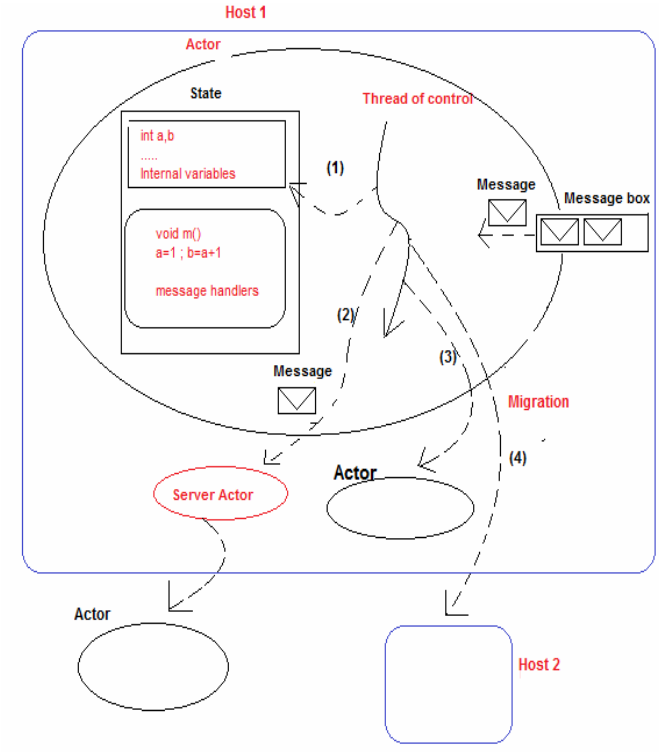
Table1: Table of needs

| | [$m_1$] | [$m_2$] | [$m_3$] |
|---|---|---|---|
| $b_1$ | $L_1$ | 0 | 0 |
| $b_2$ | 0 | $L_2$ | $L_3$ |
| $b_3$ | 0 | $L'_2$ | 0 |

The ultimate phase of our static analysis concerns the compilation of all the send of the application. At the end of this phase each **send(a,m)** is comp
iled into **send(a,m,L)**, where **L** is the number of necessary levels of the message **m** in order to be filtered and treated by the receiver actor **a**. **L** is determined by the table of needs according to the initial behavior of **a**. For example, if the initial behavior of **a** is $\mathbf{b_1}$, then **send(a,$m_1$)** is compiled into **send(a,$m_1$,$L_1$).**
Note that if the initial behavior of **a** or the structure of **m** are not known then **send(a,m)** is compiled into **send(a,m,1)**. We send one level because the rat of the message is necessary at least for the pattern-matching. We can't send more because, in this case, we haven't any more information concerning the necessary.

## IV. MOBILE IMPLEMENTATION OF LAZY COMMUNICATION STRATEGY USING MOBILE ACTOR

The actor model provides a unit of encapsulation for a thread of control along with internal state. An actor is either unblocked or blocked. It is unblocked if it is processing a message or has messages in its message box, and it is blocked otherwise. Communication between actors is purely asynchronous: non-blocking and non-First-In-First-Out (non-FIFO). However, communication is guaranteed: all messages are eventually and fairly delivered. In response to an incoming message, an actor can use its thread of control to 1) ***modify its encapsulated internal state***, 2) ***send messages to other actors***, 3) ***create actors***, or 4) ***migrate to another host***.




Picture 1

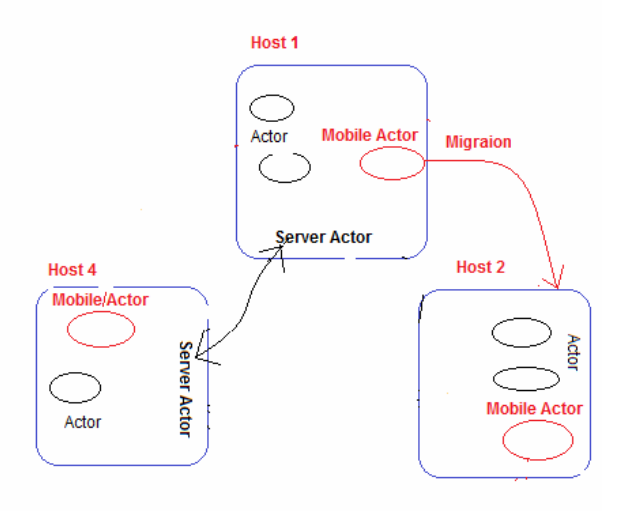
The create primitive (**create($X_0$)**) allocates a unique mail address to the newly created actor, and creates a thread which represents the computation potency of this created actor. The system makes an adequacy between the actor and its mail queue. So, the name of an actor and its mail queue address become synonymous.
Each new actor is created in the host having the minimal number of process. It's important to allow the programmer to ignore the details concerning the physical location of the actors in different processors which constitute the network of the program execution. To that effect, every site has a **server actor**. This actor manages the distribution of the actors and the communication between sites.

The mobile implementation of communication strategy witch we present in this paper is based on mobile actors. In the distributed actor system, each host has its mobile actor witch can migrate to the other hosts with the parts of messages, that must be transmitted outside.

Mobil actor is a soft entity that can migrate during the execution from host to other host. It migrates with its code, its execution state and its data. The mobility is controlled by the application not by the execution system. Le but of the migration is to reach distant data and resources, to do the treatment locally and ***to move just necessary data***.

So, mobile agent migrates to the hosts which have a lot of interactions with his host. In the case of the low interactions, this approach become less efficient, it's penalized by the size of the mobile actor. In this second case, the messages towards distant sites are addressed to the server actor of the sender site in order to treat and send them to their destination. In the receiver site, the server actor treats the external messages that arrive and transmit them to their receivers.

Using mobile actor involves the optimization of the traffic because exchanged messages become locals and discharge the network. Mobil actors can use the resources of distant hosts, so, it discharges its initial host. The size of the mobile actor adds a surcharge when it's migrating but this charge is compensated by local interactions.




Picture 2

In each host, the static analysis is done by some local calculator actors. The results of the staticanalysis are deposits in the table of need as explained above.

## CONCLUSION

We have presented a mobile implementation of lazy strategy of term communication in a distributed environment of actors. We have used a mobile actor which migrates to distant sites having massive interactions with its site. This approach

minimizes the traffic in the distributed system of actors. We are testing it on some consistent benchmarks cost concerning the execution time. This work combines into a global system for the valuation of the actor languages through a distributed virtual machine MVAD using the mobile actors.

REFERENCES:

[1] Najat. Rafalia&Jaafarbouchabaka, An extension to λ-calculus for Distributed Actor Programming. April 2013. International Journal of Management & Information Technology, Volume 3 N° 3, 2013, ISSN2278-5612.

[2] Kang Lianghan, Cao Donggang, 2012. An extension to Computing Element in Erlang for Actor Based Concurrent Programming 2012 IEEE, 15thInternational Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops..

[4] Greg Michaelson. Published at August 2011 by Dover Publications, (first published 1989).An Introduction to Functional Programming Through Lambda Calculus.

[5]By Wei-Jen Wang, DISTRIBUTED GARBAGE COLLECTION FOR LARGE-SCALE MOBILE ACTOR SYSTEMS A Thesis Submitted to the Graduate , Faculty of Rensselaer Polytechnic Institute, in Partial Fulfillment of the Requirements for the Degree of DOCTOR OF PHILOSOPHY Major Subject: Computer Science , New York, December 2006, (For Graduation Dec 2006)

[6 ] The USENIX Association, All Rights.*5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)*, San Diego, California, USA, May 3–7, 1999, JMAS: A Java-Based Mobile Actor Systemfor Distributed Parallel Computation. *Lesgand L. Burge III, Howard University, K. M. George, Oklahoma State University,* © 1999

[7] A.Elfaker, M. Pantel, P. Sallé and X. Massoutié. Octobre 1995. Vers une Machine Virtuelle pour l'évaluation des langages d'acteurs. LMO'95, Nancy pages 221-239.

[8] GulAgha,Chris Houck and RajendaPanwar. August 1991.University of Illinois at Urbana, II 61801, USA.Distributed Execution of Actor Programs Sciences Forth workshop on language and compiler for parallel computing.

[9] Henry E.Bal, Adrew S. Tanenbaum, September 1989. Department of Mathematics and Computer Sciences, Vrije University, Amsterdam, Jennifer G. Steiner, centrum voorwiskinde en Informatica, Amsterdam, the Netherlands. Programming Language for Distributed System.ACM Computing Surveys, Vol 21.

[10]Gul Agha (1986) Retrieved 2012-12-02..Actors: a Model of Concurrent Computation in Distributed-Systems. Doctoral Dissertation, Mit Press.

AUTHOR PROFILES:

**Pr. NajatRafalia,** she has obtained her doctorate in Computer Sciences at Mohammed V University, Rabat, Morocco. Currently she is a professor at IbnTofail University, Department of Computer Sciences, Kénitra, Morocco. Her research interests are in distributed systems, concurrent and parallel programming, communication and multi agent systems.

**Pr. JaafarAbouchabaka,** he has obtained two doctorates in Computer Sciences applied to mathematics at Mohammed V University, Rabat, Morocco. Currently he is a professor at IbnTofail University, Department of computer Sciences, Kénitra, Morocco. His research interests are in concurrent and parallel programming, distributed systems, and multi agent systems.