

Behavior Model of Mobile Agent Systems

Pierre Vignéras
LaBRI - Université Bordeaux 1
351, cours de la Libération
33405, Talence, France
vigner@labri.fr

February 22, 2005

Abstract

Mobile agents is a paradigm used in the mobile code area among others. It has been studied for several years and many implementations are available. Nevertheless, the technology is far from being widely accepted for many reasons. One of these reasons is the lack of behavior models of mobile agent systems preventing their theoretical study. This paper presents the mobile agent paradigm, some implementations of it, and a π -calculus model. This model leads to a new concept that we have called *active containers*. We claim it is a basic brick on which any other mobile code paradigm can be expressed.

Keywords: behavior model, mobile agent system, active container.

1 Introduction

Five years ago, the community was very excited by mobile agents. The paradigm promised to be revolutionary, the future of the Internet should have seen many agents moving from host to host to perform tasks on the behalf of their users. Many works were done on this subject, many implementations was made [1]. But, we must admit that mobile agents are far from being widely accepted. It is really hard to find a publicly available agent server on the Internet.

Many reasons can be given [2]. But we believe models of mobile agent systems are lacking. Nev-

ertheless, one work studied the infrastructure needed for the deployment of mobile agents [3] while an other focused on the patterns commonly used in mobile agents programming [4]. A structural model is proposed in [5] using UML diagrams. The only model which aims to model the behavior of mobile agent systems has been found in [6]. But we believe it is too far from real implementations. For example, it does not raise security problems related to the language used (Java for the most of them).

So we first describes the mobile agent paradigm in section 2. We then briefly present the π -calculus in section 3 used to design our model. Then our behavior model is proposed in section 4. This model leads naturally to a new *active* data structure we have called *active container* presented in section 5. We finally conclude giving some directions for future works.

2 Mobile Agent Paradigm

At least three paradigms are known to be used in mobile code computing [7]: remote evaluation (REV), code on demand (COD), and mobile agent (MA). The latter is defined by the ability of a code components to move to a host where it may continue its computation using some resources available on its destination. While in REV and COD, the focus is on the transfer of code between components; in the mobile agent paradigm, a whole computational component is

moved to a remote site, along with its state, the code it needs, and some resources required to perform the task is has been created for.

Note that the term “agent” has also a meaning in the artificial intelligence domain. So, one may think that a mobile agent contains some sort of “intelligence”. But this is not necessary, at least in our model. We define an agent as an *autonomous* and *independent* entity:

autonomous: it has the control of its own execution and does not require an interaction to complete its task;

independent: it is executed in its own thread.

Several constraints exist in mobile agent programming such as security, portability, or dynamic linkage [8]. Hence languages which provide facilities to deal with some of this issues are naturally more suited than others. Java is known to be a good language for distributed programming in general and for mobile code programming in particular [9]. Hence we will focus on mobile agent systems written in Java but we believe our study may be extended to any other language¹.

Almost every Java mobile agent system provides an event-based mechanism to provide *name resolution*. When an object move, some of its references must be modified. For example, a monitor must be released before the migration and a new one acquired on destination to prevent deadlock. References to [Input|Output]Stream instances must be modified to avoid the throws of exceptions due to their non-serializable nature. Aglets [10] API provides several methods for this purpose such as `onMigrating()` and `onMigration()` respectively invoked before and after the migration of an aglet. Voyager [11] supplies the methods `[pre|post]Departure()` and `[pre|post]Arrival()` which play almost the same roles. Grasshopper [12] has the equivalent methods `[before|after]move()` and Agent OS [13] also defines two methods `onArrival()` and `onDispatch()`. Almost any mobile agent system written in Java provides an event-based mechanism².

¹But perhaps with many more difficulties.

²Since it is impossible in Java to get back the instruction

3 π -calculus overline

To describe our model, we need a formal notation able to express entity communications and agent migrations. Several calculus for mobile processes are available [15]. We use the polyadic π -calculus [16, 17, 18] for our model.

We consider an infinite set of names $\mathcal{X} = \{x, y, \dots\}$ and the set of processes $\mathcal{Q} = \{P, Q, \dots\}$. A process may be in the following form:

$\mathbf{0}$ is the *nil* process which does nothing;

$F \equiv (\lambda \vec{x}).P$ is an *abstraction* of arity $|F| = |\vec{x}|$;

$C \equiv (\nu \vec{y})[\vec{x}]P$ with $(\vec{y} \subseteq \vec{x})$ is a *concretion* of arity $|C| = |\vec{x}|$;

$\sigma = z/y$ is a *substitution*: the syntax replacement of y by z ; $P\sigma$ is the application of this substitution to P .

$x.(\lambda \vec{y}).P = x.F$ is an *input prefix* meaning that $|\vec{y}|$ names are received along the name (*port*) x ;

$\bar{x}(\vec{y}).P = \bar{x}.C$ is an *output prefix* meaning that $|\vec{y}|$ names are sent along the name (*port*) x ;

$P + Q$ is a *sum process* that can enact either P or Q ; in particular, they cannot mutually interact;

$P|Q$ is a *parallel process* which represents the parallel execution of P and Q ; they can act independently, and may also communicate if one performs an output and the other an input along the same *port*;

$(\nu \vec{x}).P$ is a *restriction* where the $n = |\vec{x}|$ names are local to P and cannot be immediately used as *ports* for communication between P and its environment; however, they can be used for communication between components within P ;

$F \bullet C$ is a *pseudo application* defined as follow:
 $F \equiv (\lambda \vec{x}).P$, $C \equiv (\nu \vec{z})[\vec{y}]Q$ and $\vec{x} \cap \vec{z} = \emptyset$ then
 $w.F|\bar{w}.C \rightarrow F \bullet C \stackrel{\text{def}}{=} (\nu \vec{z})(P\{\vec{y}/\vec{x}\}|Q)$.

pointer, Java mobile agent systems provide only *weak migration* [14] which usually requires an event-based mechanism.

With this definition, the reduction rules are the following:

$$\text{COMM} : (\dots + x.F) | (\dots + \bar{x}.C) \rightarrow F \bullet C$$

$$\text{PAR} : \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{RES} : \frac{P \rightarrow P'}{(\nu x)P \rightarrow (\nu x)P'}$$

$$\text{STRUCT} : \frac{Q \equiv P \quad P \rightarrow P' \quad P' \equiv Q'}{Q \rightarrow Q'}$$

As we deal with mobile agent systems written in Java, we model Java code and we use the following notation:

$\{\tilde{X}\#x_1, \dots, \tilde{X}\#x_n\}$ is a set of methods names. This set is noted \tilde{X}^3 . We write $\tilde{X} \asymp \{x_1, \dots, x_n\}$ to define \tilde{X} . Finally, $\tilde{X}\#x_i = \tilde{X}\#\bar{x}_i$

For readers not quite comfortable with the π -calculus, we give a short example of the following Java code expressed in π -calculus :

```
public class MyClass{
  public int ma(int xa){
    return plus(xa,2);
  }
  public type mb(type xb){
    ...
  }
  ...
  public type mz(type xz){
    ...
  }
}
```

³The symbol # may be considered as the concatenation operator.

$$\tilde{m} \asymp \{ma, mb, \dots, mz\}$$

$$\text{MyClass} \equiv (\lambda \tilde{m}) \left\{ \begin{array}{l} \tilde{m}\#ma(\lambda xa, result). \\ \quad \overline{plus}[xa, 2, result] + \\ \tilde{m}\#mb(\lambda xb, result). \\ \quad \dots + \\ \vdots \\ \tilde{m}\#mz(\lambda xz, result). \\ \quad \dots \end{array} \right\}$$

The code:

```
MyClass o = new MyClass();
int y = plus(o.ma(3), 2);
```

is expressed by:

$$(\nu \tilde{m}) \left\{ (\text{MyClass})(\tilde{m}) \mid \right. \\ \left. (\nu y, res) \left[\tilde{m}\#\overline{ma}[3, res] \mid \overline{plus}[res, 2, y] \right] \right\}$$

Note that the access to a variable or to a method is expressed by a *port* access and thus requires a communication.

4 Proposition of a Mobile Agent System Model

Mobile agents exist only when they are *contained* into an agent server. This containment relation is really important and is the base of our model. Furthermore, the only way to contact a mobile agent is through the agent server it resides in⁴. The server which contains an agent is said to be *local* to it.

Our model has the following characteristic:

⁴This assumption is not always true since some services may be contained into the server itself (as agent in **AgentOS** for example or as external entities in **Voyager**). But from the outside point of view, the agent server is the one which perform the communication function on the behalf of its contained mobile agent.

The local server is the only entity able to invoke methods of its contained agents;

This characteristic implies that two agents cannot communicate directly. Hence, when an agent A wants to communicate with an other agent B , he must contact the local server S_B of B and ask him to invoke a method on B . Hence, an identifier id must be available in order to identify an agent in its local server. Representing an agent migration from a host S_a to a host S_b is just a matter of invalidating the identifier used in S_a and create a new one in S_b which will represent the arrived agent on S_b .

Hence, we consider a network $\mathcal{R} = S_1, \dots, S_n$ of agent systems where S_i is an agent system which contains k_i agents $A_{i,1}, \dots, A_{i,k_j}$. The running system is expressed by the π -calculus expression:

$$\begin{array}{l|l} S_1 & \{A_{1,1}|A_{1,2}|\dots|A_{1,k_1}\} \\ S_2 & \{A_{2,1}|A_{2,2}|\dots|A_{2,k_2}\} \\ \vdots & \vdots \\ S_n & \{A_{n,1}|A_{n,2}|\dots|A_{n,k_n}\} \end{array}$$

An agent server is a set of services such as *createAgent*, *sendAgent*, *receiveAgent*, *killAgent* remotely accessible and a set of agents only accessible through an identifier id . Two other names is used internally by the agent server and for agent creation and destruction: *NewAgent* and *DeleteAgent*.

Mobile agents are defined by the set of methods used by agent servers on each event of their life⁵:

onCreation(): invoked after the creation of the agent;

onMigrating(): invoked just before its migration;

onMigration(): invoked just after its migration;

onDisposing(): invoked just before its destruction.

The real code of these methods are only known by the agent and are represented by a set of abstractions:

⁵Here, the model is very inspired by the Aglets [19] system, but any other system may be modeled using a similar approach.

$\{C, R, M_o, M_i, D, Extra\}$ where C is the abstraction related to the creation of the agent taking three parameters: a “custom” argument, its id and its local server. Hence, ($|C| = 3$). Similarly, R ($|R| = 0$) represents the business logic of the agent; M_o ($|M_o| = 1$) is the code run after the agent migration, the new local server is taken in argument; M_i ($|M_i| = 1$) is the code run just before the migration, the destination server is taken in parameter; and D ($|D| = 0$) is the code related to the destruction of the agent. The *Extra* abstraction is not used by the server and represents the state of the agent and some other code which may be used by R . Each abstraction is related to a method name as given in the table 4.

Abstraction	Name
C	<i>onCreation</i>
M_i	<i>onMigrating</i>
M_o	<i>onMigration</i>
D	<i>onDisposing</i>

Table 1: Agent methods and abstraction relationship

Two other methods are defined in the agent:

migrate(): invoked for a migration request;

dispose(): invoked for a removal request.

These names are not related to abstractions because they are external, accessible by the agent and also by any other entities of the whole system which knows the id of the agent. Hence, we define:

$$\tilde{S} \asymp \{createAgent, sendAgent, receiveAgent, killAgent\}$$

$$AgentCode \widetilde{\asymp} \{C, R, M_o, M_i, D, Extra\}$$

$$\begin{aligned} \widetilde{AgentMethod} \simeq \{ & onCreation, \\ & onMigration, \\ & onMigrating, \\ & onDisposing, \\ & migrate, \\ & dispose \} \end{aligned}$$

4.1 Agent Server Model

An agent server is defined by the following abstraction:

$$\begin{aligned} Server \equiv (\lambda \tilde{S}) & \\ (\nu NewAgent, DeleteAgent) & \\ !((Agents)(\tilde{S}, NewAgent, DeleteAgent) & \\ |(Services)(\tilde{S}, NewAgent, DeleteAgent)) & \end{aligned}$$

After the creation of two names *NewAgent* and *DeleteAgent*, the *Server* abstraction applies both abstractions *Agents* and *Services*. The replication operator allows the server to wait for request indefinitely. Instantiation of a server is done by the following expression:

$$(\nu \tilde{S})(Server)(\tilde{S})$$

The *Agents* abstraction represents all the agents contained in the server and is defined as follow:

$$\begin{aligned} Agents \equiv (\lambda \tilde{S}, NewAgent, DeleteAgent). \{ & \\ NewAgent(\lambda id, AgentCode). & \\ (\nu AgentMethod) & \\ \{ (Agent)(AgentCode, AgentMethod, & \\ id, \tilde{S}) & \\ |!id[AgentCode, AgentMethod] \} + & \\ DeleteAgent(\lambda id). & \\ id(\lambda AgentCode, AgentMethod). & \\ AgentMethod\#\overline{onDisposing} \} & \end{aligned}$$

Agent creation is done through the *port NewAgent* which receives a new *id* and the agent code. The

agent is considered alive after the fresh creation of its method names *AgentMethod* and by the application of the abstraction *Agent* ($|Agent| = 4$) to the following parameters: $(AgentCode, AgentMethod, id, \tilde{S})$. Thus, the agent knows:

- the set *AgentMethod* of its method names;
- its identifier *id* on its local server;
- the methods \tilde{S} of its local server.

The creation ends with the output on the *id port* of the code and the methods of the agent. Removal of an agent is trivial: the server invokes the *onDisposing* method.

Agent server services are the set of public method names. For example, *createAgent* is the service responsible of the creation of a mobile agent in the system. It receives the agent code, creates a new *id*, and invokes the private *NewAgent* method which instantiates the abstraction representing the agent code, invokes the *onCreation* method of the new agent, and returns the fresh *id* enabling future communication with the agent. Hence, we have:

$$\begin{aligned} \tilde{S}\#createAgent(\lambda AgentCode, arg, getid). & \\ (\nu id)\overline{NewAgent}[id, AgentCode]. & \\ id(\lambda AgentCode, AgentMethod). & \\ AgentMethod\#\overline{onCreation}[arg]. & \\ \overline{getid}[id] & \end{aligned}$$

Similarly, the sending of an agent by a server \tilde{S} to a server \tilde{S}' needs several steps:

- the reception of the *id* of the agent to send by the method $\tilde{S}\#sendAgent$;
- the invocation of the method *onMigrating* of the agent to migrate;
- the invocation of the method $\tilde{S}'\#receiveAgent$ of the destination server;
- the return of the new *id* of the agent into its new local destination server.

Hence we have:

$$\begin{aligned} & \widetilde{S} \# \overline{\text{sendAgent}}(\lambda id, \widetilde{S}', \overline{\text{getNewid}}). \\ & \quad id(\lambda \overline{\text{AgentCode}}, \overline{\text{AgentMethod}}). \\ & \quad \overline{\text{AgentMethod}} \# \overline{\text{onMigrating}}[\widetilde{S}']. \\ & (\nu \overline{\text{getid}}) \widetilde{S}' \# \overline{\text{receiveAgent}}[\overline{\text{AgentCode}}, \overline{\text{getid}}]. \\ & \quad \overline{\text{getid}}(\lambda \overline{\text{Newid}}). \\ & \quad \overline{\text{getNewid}}[\overline{\text{Newid}}] \end{aligned}$$

The two other methods *receiveAgent* and *killAgent* are quite identical. The complete model of the agent server services is:

$$\begin{aligned} \text{Services} \equiv (\lambda \widetilde{S}). \left\{ \right. \\ & \quad \widetilde{S} \# \overline{\text{createAgent}}(\lambda \overline{\text{AgentCode}}, \overline{\text{arg}}, \overline{\text{getid}}). \\ & \quad (\nu id) \overline{\text{NewAgent}}[id, \overline{\text{AgentCode}}]. \\ & \quad \quad id(\lambda \overline{\text{AgentCode}}, \overline{\text{AgentMethod}}). \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{onCreation}}[\overline{\text{arg}}]. \\ & \quad \quad \overline{\text{getid}}[id] \\ + & \quad \widetilde{S} \# \overline{\text{sendAgent}}(\lambda id, \widetilde{S}', \overline{\text{getNewid}}). \\ & \quad id(\lambda \overline{\text{AgentCode}}, \overline{\text{AgentMethod}}). \\ & \quad \overline{\text{AgentMethod}} \# \overline{\text{onMigrating}}[\widetilde{S}']. \\ & \quad (\nu \overline{\text{getid}}) \widetilde{S}' \# \overline{\text{receiveAgent}}[\overline{\text{AgentCode}}, \\ & \quad \quad \overline{\text{getid}}]. \\ & \quad \quad \overline{\text{getid}}(\lambda \overline{\text{Newid}}). \\ & \quad \quad \overline{\text{getNewid}}[\overline{\text{Newid}}] \\ + & \quad \widetilde{S} \# \overline{\text{receiveAgent}}(\lambda \overline{\text{AgentCode}}, \overline{\text{getid}}). \\ & \quad (\nu id) \widetilde{S} \# \overline{\text{NewAgent}}[id, \overline{\text{AgentCode}}]. \\ & \quad \quad id(\lambda \overline{\text{AgentCode}}, \overline{\text{AgentMethod}}). \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{onMigration}}. \\ & \quad \quad \overline{\text{getid}}[id] \\ + & \quad \widetilde{S} \# \overline{\text{killAgent}}(\lambda id). \widetilde{S} \# \overline{\text{DeleteAgent}}[id] \left. \right\} \end{aligned}$$

4.2 Mobile Agent Model

Agent may want to communicate with its local server to request a migration for example. Other entities in the whole system may also want to communicate with it through its *id*. Its method *onCreation*, *onDisposing* and *dispose* must be called only once

during all the life of the agent whereas *onMigrating*, *onMigration* and *migrate* may be invoked many times. So the model of the agent is as the following:

$$\begin{aligned} \text{Agent} \equiv (\lambda \overline{\text{AgentCode}}, \overline{\text{AgentMethod}}, id, \widetilde{S}) \left\{ \right. \\ & \quad \left\{ (\nu \text{run}) [\right. \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{onCreation}}(\lambda \overline{\text{arg}}). \\ & \quad \quad (\overline{\text{AgentCode}} \# C)(\overline{\text{arg}}, id, \widetilde{S}). \overline{\text{run}} + \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{onMigration}}. \\ & \quad \quad (\overline{\text{AgentCode}} \# M_o)(\widetilde{S}). \overline{\text{run}} \\ & \quad \quad \left. \right] | \text{run}. (\overline{\text{AgentCode}} \# R) \left\} \\ & \quad | [\overline{\text{AgentMethod}} \# \overline{\text{onMigrating}}(\lambda \widetilde{S}'). \\ & \quad \quad (\overline{\text{AgentCode}} \# M_i)(\widetilde{S}') + \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{onDisposing}}. \\ & \quad \quad (\overline{\text{AgentCode}} \# D)] \\ & \quad | [\overline{\text{AgentMethod}} \# \overline{\text{migrate}}(\lambda \widetilde{S}'). \\ & \quad \quad \widetilde{S} \# \overline{\text{sendAgent}}[id, \widetilde{S}'] + \\ & \quad \quad \overline{\text{AgentMethod}} \# \overline{\text{dispose}}. \\ & \quad \quad \widetilde{S} \# \overline{\text{killAgent}}[id] \left. \right\} \end{aligned}$$

The method *onCreation* receives a special argument⁶ which represents the required data needed for the instantiation of the mobile agents. Then the abstraction *C* is applied and the process *R* is executed in parallel when a message is sent through the *port run* ensuring sequential execution of *C* followed by *R*. Note how a call to *onMigrating* eliminate the call to *onDisposing* using the + operator.

4.3 Studying Our Model

4.3.1 Agent Control

Once created, the agent may run in *R* almost anything. Deny of service is straightforward in agent

⁶This argument differs for each agent and the model needs the use of the polymorphic π -calculus [20] which is beyond the scope of this document.

systems written in Java. For example, the instruction:

```
while(true);
```

monopolize the processor. Moreover, it is not possible to interrupt an agent since the agent code is run in its own thread, and that interrupting a thread is really a big challenge⁷. Static analysis may help, but with a cost that prevent the mobile agent paradigm from being interesting since it can usually be replaced by a more “traditional” one [22].

Furthermore, abstractions C , M_o , M_i and D can also run “evil” code. For example, if $C \equiv P.(R)$, the (R) process will run before the end of C . This problem appears in many mobile agent system implementations and is related to the event model.

4.3.2 Agent Destruction

The π -calculus does not provide term deletion. Since we do not know how many agents will communicate with a given agent, the term $\overline{id}[AgentCode, AgentMethod]$ is replicated *ad infinitum* using the “!” operator. Hence, this term cannot be deleted⁸. Worst, the R abstraction for example, unknown from the server may contain several replications. Terms may accumulate in the expression through processing of agent creation. This problem is actually visible into mobile agent system implementations [23].

4.4 Summary

The model we propose reflect current implementations and we have seen many limitations. Other problems can be identified using such a model [23]. All are found in almost any implementation of a mobile agent system written in Java.

But as we will see, this model leads naturally to a new concept.

⁷See [21] for details.

⁸Nevertheless, if a call through id is no more possible, then the original process can be simulated with a process which does not contain i as a name (garbage collector).

5 Active Container: the Underlying Paradigm

In our mobile agent system model, servers play a central role in the communication mechanism (*cf.* section 4). Hence, for identification purpose, an agent identifier must allow:

- the finding of the server who *contains* a given agent;
- the identification of the given agent on this server.

Hence, a sort of table must be used in each agent server for this purpose. If this table maps *bijectionally* a key with an agent hosted by a server, then an agent is identified uniquely in a mobile agent system with:

- the unique identifier of a given agent server;
- the agent key in this agent server.

5.1 Container Definition

From a mobile agent system point of view, an agent migration can be replaced by two basics operations:

1. removing the agent from its current hosted server;
2. inserting the agent into its destination server.

Defining an agent server as a container with the following interface may be sufficient to express migration:

```
void put(Object key, Agent agent);
void remove(Object key);
Agent get(Object key);
```

Methods `put()` and `remove()` are self explanatory. The `get()` method returns a copy of an agent.

5.2 Agent Migration

An agent migration is thus easily expressed using a container API. Consider two agent server `s1` and `s2`. The following instructions express an agent migration from `s1` to `s2`:

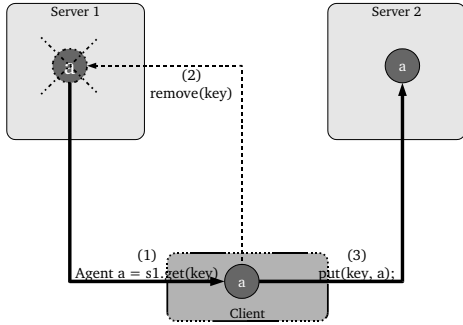


Figure 1: A supplementary migration is involved by the use of the `get()` method.

```

Agent a = s1.get(key);
s1.remove(key);
s2.put(key, a);

```

Note that a supplementary migration occurs in this code: in the first line the client gets the agent code involving a migration as shown by the figure 1. In fact, in a *proactive* mobile agent systems, it is the agent which decides of its own migration. In this case, the previous migration can be written:

```

s1.remove(myKey);
s2.put(myKey, this);

```

Hence, the container interface allows migration expression. But our system must support inter-agent communication to be useful.

5.3 Container as an *Active Data Structure*

As defined in section 4, communication between agents must pass through the agent server. For this purpose, a new method must be available in the container API. We define an *active container* as a container – as defined in section 5.1 – able to invoke methods of its stored object. The following method is thus provided:

```

void call(Object key,
          Method m,
          Object[] args,
          Result r);

```

This method invokes the specified method `m` of the object maps to `key` in the container with the argument `args` and returns the result in the object `r`. The method turns our container into an *active* data structure. Furthermore, we specify that the method `m` must be invoked *asynchronously* ensuring that agents are *autonomous* and *independent* (cf. section 2). Hence, if an agent `a1` wants to communicate with another agent, say `a2`, it has to know the active container who is storing `a2`, the key of `a2` and the method it wants to invoke⁹.

5.4 Active Container Model

In π -calculus, names play the role of keys used in the mapping of our active container. Hence, we define:

$$\widetilde{AC} \asymp \{put\}$$

While an object has not been inserted into the active container, other methods do not have to exist. An active container is then defined as:

$$\begin{aligned}
ActiveContainer \equiv & (\lambda put). \\
& ! \left\{ put(\lambda F, \tilde{O}, handles).(F)(\tilde{O}) \right. \\
& \quad \left. | (\nu get, call, remove) \right. \\
& \quad \quad \overline{handles}[get, call, remove]. \\
& \quad \left[\overline{!get}[F, \tilde{O}] \right. \\
& \quad \quad \left. | !call(\lambda m, args, result). \right. \\
& \quad \quad (\nu future) \overline{result}[future]. \\
& \quad \quad (\nu res) (\overline{m}[args, res] \\
& \quad \quad \quad \left. | res(\lambda val). \overline{future}[val])) \right] \\
& \quad + remove \\
& \left. \right\}
\end{aligned}$$

Even if problems related to objects destruction (cf. section 4.3) does not appear directly in this model, they are still there: after the reception of an empty message on the *remove* name, services *get*

⁹Method invocation is traditionally abstracted to message passing paradigm.

and *remove* are unavailable. But, a previous invocation of *call* may have created many process able to communicate directly without the participation of the container. In fact, we consider this as a feature, named *Multi-Protocols Stored Objects (MPSO)* and has an application in security [24].

Note that once an object has been inserted with the *put()* method, the name *handles* is used to further invoke the method *get*, *remove* and *call* of the container. Note also how the method *call* is made asynchronous using a name *future* which must be used to handle the result.

5.5 Mobile Agent System Simulation

It seems possible to simulate a mobile agent system using the active container concept:

$$\tilde{S} \simeq \{ \text{createAgent}, \\ \text{sendAgent}, \\ \text{receiveAgent}, \\ \text{killAgent} \}$$

Considering the set:

$$\tilde{A} \simeq \{ \text{onCreation}, C, \\ \text{onMigration}, M_o, \\ \text{onMigrating}, M_i, \\ \text{onDisposing}, D, \\ \text{run}, R \}$$

Our agent server may be written:

$$\begin{aligned} \text{Server} &\equiv (\lambda \tilde{S}) \\ &(\nu \tilde{AC})(\text{ActiveContainer})(\tilde{AC}) \\ &|(\text{Services})(\tilde{S}, \tilde{AC}) \\ \\ \text{Services} &\equiv (\lambda \tilde{S}, \tilde{AC})! \left\{ \begin{aligned} &\tilde{S}\#\text{createAgent}(\lambda \tilde{A}, \text{arg}, \text{returnid}). \\ &(\nu \text{handles})(\nu \text{id})\tilde{AC}\#\overline{\text{put}}[\text{Agent}, \tilde{A}, \\ &\hspace{10em} \text{handles}]. \\ &\text{handles}(\lambda \text{get}, \text{call}, \text{remove}). \\ &(\nu \text{result})\overline{\text{call}}[\tilde{A}\#\text{onCreation}, \\ &\hspace{10em} \tilde{S}, \text{arg}, \text{result}]. \\ &\text{result}(\lambda \text{future}).\text{future}(\lambda \text{val}). \\ &(\nu \text{dummy})\overline{\text{call}}[\tilde{A}\#\text{run}, \mathbf{0}, \text{dummy}] | \\ &\overline{\text{returnid}}[\text{id}] | \\ &!\text{id}[\text{get}, \text{call}, \text{remove}] \end{aligned} \right. \\ &+ \\ &\tilde{S}\#\text{sendAgent}(\lambda \text{id}, \tilde{S}', \text{returnNewid}). \\ &\text{id}(\lambda \text{get}, \text{call}, \text{remove}). \\ &(\nu \text{result})\overline{\text{call}}[\tilde{A}\#\text{onMigrating}, \\ &\hspace{10em} \tilde{S}', \text{result}]. \\ &\text{result}(\lambda \text{future}).\text{future}(\lambda \text{val}). \\ &\text{get}(\lambda \text{Agent}, \tilde{A}). \\ &\overline{\text{remove}}. \\ &\tilde{S}\#\text{receiveAgent}(\tilde{A}, \text{returnNewid}) \\ &+ \\ &\tilde{S}\#\text{receiveAgent}(\lambda \tilde{A}, \text{returnid}). \\ &(\nu \text{handles})(\nu \text{id})\tilde{AC}\#\overline{\text{put}}[\text{Agent}, \\ &\hspace{10em} \tilde{A}, \text{handles}]. \\ &\text{handles}(\lambda \text{get}, \text{call}, \text{remove}). \\ &(\nu \text{result})\overline{\text{call}}[\tilde{A}\#\text{onMigration}, \\ &\hspace{10em} \tilde{S}, \text{result}]. \\ &\text{result}(\lambda \text{future}).\text{future}(\lambda \text{val}). \\ &(\nu \text{dummy})\overline{\text{call}}[\tilde{A}\#\text{run}, \mathbf{0}, \text{dummy}] | \\ &\overline{\text{returnid}}[\text{id}] | \\ &!\text{id}[\text{get}, \text{call}, \text{remove}] \\ &+ \\ &\tilde{S}\#\text{killAgent}(\lambda \text{id}). \\ &\text{id}(\lambda \text{get}, \text{call}, \text{remove}). \\ &(\nu \text{result})\overline{\text{call}}[\tilde{A}\#\text{onDisposing}, \tilde{S}, \text{result}]. \\ &\text{result}(\lambda \text{future}).\text{future}(\lambda \text{val}). \\ &\overline{\text{remove}} \end{aligned}$$

Note the use of the name *handles* to retrieve the names *get*, *remove* and *call* mapped to our object. Note also the expression:

$$result(\lambda future).future(\lambda val)$$

which allows the waiting of the asynchronous invocation of *call*. After the creation of an agent, *i.e.*, after the insertion in the container of the *Agent* abstraction and of the vector of names \tilde{A} , the agent identifier allows the getting back of the names *call*, *get* and *remove*. The sending of an agent, is made by retrieving its names, invoking its *onMigrating* method synchronously, receiving a copy, removing the agent from the container and sending it to the remote server.

The agent may be written:

$$\begin{aligned}
Agent \equiv & (\lambda id, \tilde{A}). \\
& (1) \quad (\nu getCurrentServer) \left\{ \right. \\
& (2) \quad (\nu setCurrentServer) \left[\right. \\
& (3) \quad \left[\overline{setCurrentServer}(\lambda \tilde{S}). \right. \\
& (4) \quad \left. \overline{getCurrentServer}[\tilde{S}] \right] \mid \\
& \left[\overline{\tilde{A}\#onCreation}(\lambda \tilde{S}, arg). \right. \\
& \quad \left. \overline{setCurrentServer}[\tilde{S}].(\tilde{A}\#C)(arg) + \right. \\
& \quad \left. \tilde{A}\#onMigration(\lambda \tilde{S}). \right. \\
& \quad \left. \overline{setCurrentServer}[\tilde{S}].(\tilde{A}\#Mo)(\tilde{S}) \right] \mid \\
& \tilde{A}\#run.(\tilde{A}\#R) \mid \\
& \left[\overline{\tilde{A}\#onMigrating}(\lambda \tilde{S}).(\tilde{A}\#Mi)(\tilde{S}) + \right. \\
& \quad \left. \tilde{A}\#onDisposing. \right. \\
& \quad \left. \overline{setCurrentServer}[\mathbf{0}].(\tilde{A}\#D) \right] \mid \\
& \left[\overline{\tilde{A}\#migrate}(\lambda \tilde{S}').getCurrentServer(\lambda \tilde{S}). \right. \\
& \quad \left. \tilde{S}\#\overline{send}[id, \tilde{S}'] + \right. \\
& \quad \left. \tilde{A}\#dispose. \right. \\
& \quad \left. \overline{getCurrentServer}(\lambda \tilde{S}).\tilde{S}\#\overline{kill}[id] \right] \\
& \left. \right\}
\end{aligned}$$

Lines (1).. (4) model a single access variable. Methods *setCurrentServer* and *getCurrentServer* are self explanatory. The latter allows agent to request its own migration *via* its *migrate* method.

Intuitively, it seems possible to simulate formerly a mobile agent system with an active container model. A bisimulation between the two mobile agent system models given in this article must be found. For this purpose, the π -calculus *sort* system must be used to resolve the polymorphism problem we raise in section 4.2. The work of Tom Melham [25] may ease the work using the proof system HOL [26].

6 Future Works and Conclusion

Our contribution in this paper is both a new behavior model of mobile agent systems in π -calculus, and the active container abstraction which seems to be an underlying paradigm on which many other things can be made [27, 24, 28]. It seems to be a good abstraction for distributed and/or parallel programming. As such, we propose the Mandala framework [29] which contains the JACOb [30] Java API, where JACOb stands for *Java Active Container of Objects*. With this framework one can use active containers to develop distributed applications. As a proof of concept, we have developed such an application called DJFractal [31].

Furthermore, observing the current situation, many actors in the community have change their interest from mobile agent systems to distributed execution framework¹⁰. Furthermore, 54% of the links referenced on the Mobile Agent List [1] are invalid. One of the reason [2] may be the fact, given in this article, that mobile agent systems are actually based on the active container abstraction. This abstraction ease the programming of distributed applications as shown by the adoption of the container concept by some standards: J2EE and JavaSpace for example. Hence, using the mobile agent paradigm may be perceived as no more justified.

¹⁰Following the three standards J2EE, .NET and CORBA.

References

- [1] The mole team. The mobile agent list. Web, 1999.
<http://mole.informatik.uni-stuttgart.de/>.
- [2] Pierre Vignéras. *Section 4.2 – Intérêt*, pages 42 – 44. In [23], 8 novembre 2004. Rapporteurs : Doug Lea, Françoise Baude, Michel Riveill. Jury : Françoise Baude, Serge Chaumette, Olivier Coulaud, Mohamed Mosbah, Alexis Moussine-Pouchkine, Michel Riveill.
- [3] Anselm Lingnau and Oswald Drobnik. An Infrastructure for Mobile Agents: Requirements and Architecture. In *Proceedings of the 13th DIS Workshop*, Orlando, FL, USA, 1995.
- [4] Yariv Aridor and Danny B. Lange. Agent design patterns: elements of agent application design. In *Proceedings of the second international conference on Autonomous agents*, pages 108–115. ACM Press, 1998.
- [5] Alberto Silva and Jose Delgado. The agent pattern for mobile agent systems. In *European Conference on Pattern Languages of Programming and Computing, EuroPLoP*, 1998.
- [6] Calvin Shen and Larry T. Chen., 1998. "UCI Undergraduate Research Journal".
- [7] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
- [8] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. Analysing mobile code languages. In *Second International ECOOP Workshop on Mobile Object Systems*, Linz, Austria, July 1996.
- [9] Horvat Damir, Cvetkovic Dragana, Milutinovic Veljko, Kocovic Petar, and Kovacevic Vlada. Mobile agents and Java mobile agents toolkits. In *33rd Hawaii International Conference on System Sciences*, volume 8, page 8029, Maui, Hawaii, January 2000. IEEE.
- [10] IBM Corporation. Aglets home page, January 2001.
<http://www.tr1.ibm.co.jp/aglets/> et aussi <http://aglets.sourceforge.net/>.
- [11] Recursion Software (purchased from ObjectSpace). Voyager home page, Octobre 2003.
<http://www.recursionsw.com/products/-voyager/>.
- [12] Object Management Group. Grasshopper home page, octobre 2003.
<http://www.grasshopper.de/>.
- [13] Nikhil Kothari. AgentOS - A Java based mobile agent system. ICS Honors Project Final Report.
- [14] Antonio Carzaniga, Gian Pietro Picco, and Giovanni Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, 1997.
- [15] Uwe Nestmann. Links on calculi for mobile processes, January 2001.
<http://lampwww.epfl.ch/mobility/>.
- [16] Robin Milner. The polyadic π -calculus : a tutorial. Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, octobre 1991.
- [17] David Walker Robin Milner, Joachim Parrow. A calculus of mobile processes (parts I and II). Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, juin 1989.
- [18] Robin Milner. *Communicating and Mobile Systems – The Pi Calculus*. Cambridge University Press, June 1999. ISBN:0521658691.
- [19] Lange B. Danny and Oshima Mitsuru. *Programming and Deploying Mobile Agents with Java*, chapter Mobile Agents With Java: The Aglets API. 1998.

- [20] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic π -calculus. In *Principles of Programming Languages (POPL)*, 1997. Full version available as INRIA-Sophia Antipolis Rapport de Recherche No. 3042 and as Indiana University Computer Science Technical Report 468.
- [21] Sun microsystem. Why are `Thread.stop()`, `Thread.suspend()`, `Thread.resume()` and `Runtime.runFinalizersOnExit()` deprecated? Web. <http://java.sun.com/products/~jdk/1.2/docs/guide/misc/~threadPrimitiveDeprecation.html>.
- [22] Colin G. Harrison, David M. Chess, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? Technical report, T. J. Watson Research Center, Yorktown Heights, New York, 1995.
- [23] Pierre Vignéras. *Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones*. PhD thesis, Université de Bordeaux 1, LaBRI, 8 novembre 2004. Rapporteurs : Doug Lea, Françoise Baude, Michel Riveill. Jury : Françoise Baude, Serge Chaumette, Olivier Coulaud, Mohamed Mosbah, Alexis Moussine-Pouchkine, Michel Riveill.
- [24] Serge Chaumette and Pierre Vignéras. Extensible and customizable just-in-time-security (JITS) management of client-server communication in Java. In G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors, *Advances in Parallel Computing*, Elsevier, Netherlands, 2003. Parallel Computing: Software Technology, Algorithms, Architectures and Applications. <http://rparco.urz.tu-dresden.de/~Parco2003/up/1p-abstract-111.pdf>.
- [25] T. F. Melham. A mechanized theory of the π -calculus in HOL. Technical report, Département d'informatique 'a l' universit'e de Glasgow, Ecosse, 1992.
- [26] T.F. Melham. *Introduction to the HOL theorem prover*. University of Cambridge, Computer Laboratory, Cambridge, England, 1990.
- [27] Serge Chaumette and Pierre Vignéras. A framework for seamlessly making object oriented applications distributed. In *International Conference on Parallel Computing: PARCO'03*, 2003. Dresde, Germany.
- [28] Pierre Vignéras. Jacob : a software framework to support the development of e-services, and its comparison to enterprise javabeans. In *Proceedings of International Workshop on Performance-Oriented Application Development for Distributed Architectures (PADDA). Perspectives for Commercial and Scientific Environments*, pages 11–12, April 19-20 2001. Munchen.
- [29] Pierre Vignéras. Mandala. Web page, August 2004. <http://mandala.sf.net/>.
- [30] Serge Chaumette and Pierre Vignéras. Active containers: an alternative approach to mobile agents systems. Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98., 1998. Santa Fe, NM, USA. Poster.
- [31] Pierre Vignéras. DJFractal. Web page, August 2004. <http://djfractal.sf.net/>.