Behavior Model of Mobile Agent Systems

Serge Chaumette and Pierre Vignéras
LaBRI, Université Bordeaux 1
351, cours de la Libération
33405, Talence, France
{serge.chaumette, pierre.vigneras}@labri.fr

Abstract

The paradigm of mobile agents is used among others in the mobile code area. 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 that prevents 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 that it is a basic brick on which any other mobile code paradigm can be built.*

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

1 Introduction

Five years ago, the community was very excited by the paradigm of mobile agents. It was very promising, and people thought that the future of the Internet should give rise to many agents moving from host to host to perform tasks on behalf of their users. Many projects were carried out on this subject, many implementations were achieved [1]. But one 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.

Among the many reasons that can be given [2], we believe that models of mobile agent systems are missing. The only model the aim of which is to describe the behavior of mobile agent systems has been found in [3]. But we believe it is too far from real implementations. For example, it does not raise security problems related to the language being used.

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

2 The Paradigm of Mobile Agents

At least three paradigms are commonly used in the domain of mobile code [4]: remote evaluation (REV), code on demand (COD), and mobile agents (MA). The latter is defined by the ability of a code component to move to a host where it may continue its computation using resources available at its destination. 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" is also used in the domain of artificial intelligence. 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: it is autonomous because it has the control of its own execution and does not require any interaction to complete its task; it is independent because it is executed in its own thread.

Several constraints exist in mobile agent programming such as security, portability, or dynamic linkage [5]. Hence languages that provide facilities to deal with some of these 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 [6]. Therefore we will focus on mobile agent systems written in Java but we believe our study may be extended to any other language[1].

Almost every Java mobile agent system has an event-based mechanism to provide *name resolution*. When an object moves, some of its references must be modified. For example, a monitor must be released before the migration and a new one acquired at the destination to prevent deadlocks. References to `[Input|Output]Stream` instances must be modified to avoid exceptions from being thrown due to their non-serializable nature. The Aglets [7] API provides several methods for this purpose such as `onMigrating()` and

[1] Probably with many more difficulties.

`onMigration()` which are respectively invoked before and after the migration of an aglet. Voyager [8] supplies the methods `[pre|post]Departure()` and `[pre|post]Arrival()` that play almost the same roles. Grasshopper [9] has the equivalent methods `[before|after]move()` and Agent OS [10] also defines two methods `onArrival()` and `onDispatch()`. Almost any mobile agent system written in Java provides such an event-based mechanism[2].

3 Overview of the π-calculus

To describe our model, we need a formal notation to express both communication and migration. Several calculus for mobile processes are available [12]. We use the polyadic π-calculus [13, 14, 15].

We consider an infinite set of names $\mathcal{X} = \{x, y, \ldots\}$ and the set of processes $\mathcal{Q} = \{P, Q, \ldots\}$. A process can have the following forms:

- $\mathbf{0}$ is the *nil* process that 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 syntactic 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;
- $\overline{x}(\vec{y}).P = \overline{x}.C$ is an *output prefix* meaning that $|\vec{y}|$ names are sent along the name (*port*) x;
- $P + Q$ is a *sum process* that can behave either like P or Q; in particular, they cannot mutually interact;
- $P|Q$ is a *parallel process* that 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 $w.F|\overline{w}.C \rightarrow F \bullet C \stackrel{\text{def}}{=} (\nu\ \vec{z})(P\{\vec{y}/\vec{x}\}|Q)$.

With this definition, the reduction rules are the following:

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

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

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

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

To model Java code we use the following notation:

$\{\widetilde{X}\#x_1, \ldots, \widetilde{X}\#x_n\}$ is a set of method names. This set is noted $\widetilde{X}$[3]. We write $\widetilde{X} \asymp \{x_1, \ldots, x_n\}$ to define $\widetilde{X}$. Finally, $\overline{\widetilde{X}\#x_i} = \widetilde{X}\#\overline{x_i}$

For readers not quite comfortable with the π-calculus, we give a short example of Java code and its π-calculus expression:

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

$$\widetilde{m} \asymp \{ma, mb, \ldots, mz\}$$

$$MyClass \equiv (\lambda\ \widetilde{m})\Big\{ \widetilde{m}\#ma(\lambda\ xa, result).\ \overline{plus}[xa, 2, result] + \widetilde{m}\#mb(\lambda\ xb, result).\ \cdots + \vdots \ \widetilde{m}\#mz(\lambda\ xz, result).\ \ldots \Big\}$$

The code:

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

is expressed by:

$$(\nu\ \widetilde{m})\Big\{ (MyClass)(\widetilde{m}) \mid (\nu\ y, res)\ \Big[\ \widetilde{m}\#\overline{ma}[3, res] \mid \overline{plus}[res, 2, y]\Big]\Big\}$$

Note that variables and methods are accessed through a *port* what requires communication.

[2] Since it is impossible in Java to restore the instruction pointer, Java mobile agent systems only provide *weak migration* [11] which usually requires an event-based mechanism.

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

4 Proposition of a Mobile Agent System Model

Mobile agents exist only when they are *contained* within an agent server. This containment relation is really important and is the basis of our model. Furthermore, the only way to contact a mobile agent is through the agent server it resides in[4]. The server that contains an agent is called its *local* server.

Our model has the characteristic that *the local server is the only entity able to invoke the methods of its contained agents;* This characteristic implies that two agents cannot communicate directly. Hence, when an agent A wants to communicate with an agent B, it must contact the local server S_B of B and ask it 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 creating a new one in S_b which will represent the agent on S_b.

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

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

An agent server is a set of remotely accessible services such as $createAgent$, $sendAgent, receiveAgent, killAgent$ and a set of agents only accessible through an identifier *id*. Two other names are 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 life cycle[5]:
- `onCreation()`: invoked after the agent is created;
- `onMigrating()`: invoked just before its migration;
- `onMigration()`: invoked just after its migration;
- `onDisposing()`: invoked just before its destruction.

The real code of these methods is only known by the agent and is represented by a set of abstractions: $\{C, R, M_o, M_i, D, Extra\}$ where C is the abstraction related to the creation of the agent; it has three parameters: a "custom" argument, its *id* and its local server. Hence, ($|C| = 3$). R ($|R| = 0$) represents the business logic of the agent; M_o ($|M_o| = 1$) is the code that is executed after the agent migration, the new local server is taken as argument; M_i ($|M_i| = 1$) is the code run just before the migration, the destination server is taken as parameter; and D ($|D| = 0$) is the code that deals with 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	$onCreation$
M_i	$onMigrating$
M_o	$onMigration$
D	$onDisposing$

Table 1: Agent methods and abstraction relationship

Two other methods are defined in the agent: `migrate()` that is invoked for a migration request and `dispose()` that is 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 entity of the whole system provided it knows the *id* of the agent. Hence, we define:

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

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

$$\widetilde{AgentMethod} \asymp \{onCreation, onMigration, onMigrating, onDisposing, migrate, dispose\}$$

4.1 Agent Server Model

An agent server is defined by the following abstraction:

$$\begin{array}{l} Server \equiv (\lambda\ \widetilde{S}) \\ \quad (\nu\ NewAgent, DeleteAgent) \\ \quad\quad !((Agents)(\widetilde{S}, NewAgent, DeleteAgent) \\ \quad\quad\quad |(Services)(\widetilde{S}, NewAgent, DeleteAgent)) \end{array}$$

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 requests indefinitely. A server is instantiated by the following expression:

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

The $Agents$ abstraction represents all the agents

[4]This assumption is not always true since some services may be contained in the server itself (as agents in `AgentOS` for example or as external entities in Voyager). But seen from the outside, the agent server is the one that performs the communication function on behalf of its contained mobile agent.

[5]Here, the model is very inspired by the Aglets [16] system, but any other system may be modeled using a similar approach.

contained in the server and is defined as follows:

$$Agents \equiv (\lambda\ \widetilde{S}, NewAgent, DeleteAgent).\Big\{$$
$$NewAgent(\lambda\ id, \widetilde{AgentCode}).$$
$$(\nu\ \widetilde{AgentMethod})$$
$$\Big\{(Agent)(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S})$$
$$|!\overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}]\Big\}+$$
$$DeleteAgent(\lambda\ id).$$
$$id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}).$$
$$\widetilde{AgentMethod}\#\overline{onDisposing}\Big\}$$

Agent creation is done through the *port* $NewAgent$ which receives a new id and the agent code. The agent is considered alive after the creation of its fresh method names $\widetilde{AgentMethod}$ and by the application of the abstraction $Agent$ ($|Agent| = 4$) to the following parameters: $(\widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S})$
Thus, the agent knows: the set $\widetilde{AgentMethod}$ of its method names; its identifier id on its local server; the methods $\widetilde{S}$ of its local server.

The creation ends with the output on the *port* id of the code and the methods of the agent, what will make it possible to use it. Removal of an agent is straightforward: 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 a fresh id enabling future communication with the agent. Hence, we have:

$$\widetilde{S}\#createAgent(\lambda\ \widetilde{AgentCode}, arg, getid).$$
$$(\nu\ id)\overline{\widetilde{NewAgent}}[id, \widetilde{AgentCode}].$$
$$id(\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}).$$
$$\widetilde{AgentMethod}\#\overline{onCreation}[arg].$$
$$\overline{getid}[id]$$

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

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

Hence we have:

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

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

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

4.2 Mobile Agent Model

An agent may want to communicate with its local server for instance to request a migration. 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. The model of the agent is therefore as follows:

$$Agent \equiv (\lambda\ \widetilde{AgentCode}, \widetilde{AgentMethod}, id, \widetilde{S})\Big\{$$
$$\Big\{(\nu\ run)[$$
$$\widetilde{AgentMethod}\#onCreation(\lambda\ arg).$$
$$(\widetilde{AgentCode}\#C)(arg, id, \widetilde{S}).\overline{run}\quad +$$
$$\widetilde{AgentMethod}\#onMigration.$$
$$(\widetilde{AgentCode}\#M_o)(\widetilde{S}).\overline{run}$$
$$]\ |run.(\widetilde{AgentCode}\#R)\Big\}$$

$$
\left.
\begin{array}{l}
|\big[\widetilde{AgentMethod}\#onMigrating(\lambda\ \widetilde{S'}).\\
\quad (\widetilde{AgentCode}\#M_i)(\widetilde{S'}) +\\
\quad \widetilde{AgentMethod}\#onDisposing.\\
\quad (\widetilde{AgentCode}\#D)\big]\\
|\big[\widetilde{AgentMethod}\#migrate(\lambda\ \widetilde{S'}).\\
\quad \widetilde{S}\#\overline{sendAgent}[id, \widetilde{S'}] +\\
\quad \widetilde{AgentMethod}\#dispose.\\
\quad \widetilde{S}\#\overline{killAgent}[id]\big]
\end{array}
\right\}
$$

The method *onCreation* receives a special argument[6] 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* eliminates the call to *onDisposing* using the + operator.

4.3 Studying Our Model

Agent Control. Once created, the agent may run in R almost anything. Denial of service is straightforward to achieve in agent systems written in Java. For example, the instruction `while(true);` monopolizes 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 challenge[7]. Static analysis may help, but its cost prevents the mobile agent paradigm from being interesting since it can usually be replaced by a more "traditional" one [19].

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 due to the event model.

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 $\overline{id}[\widetilde{AgentCode}, \widetilde{AgentMethod}]$ is replicated *ad infinitum* using the "!" operator. Hence, this term cannot be deleted[8]. 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 present in most if not all mobile agent system implementations [2].

4.4 Summary

The model we propose reflects current implementations and we have seen many limitations. Other problems can be identified using such a model [2]. All are found in almost any implementation of a mobile agent system written in Java.

[6]This argument differs for each agent and the model needs the use of the polymorphic π-calculus [17] which is beyond the scope of this document.

[7]See [18] for details.

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

5 *Active Containers:* the Underlying Paradigm

In our mobile agent system model, servers play a central role in the communication mechanism (*cf.* section 4). Hence an agent identifier must allow to find the server that *contains* a given agent and to identify the given agent on this server. A sort of table must then be used in each agent server for this purpose. If this table *bijectively* associates 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 and the agent key in this agent server.

5.1 Container Definition

From the point of view of a mobile agent system, the migration of an agent can be achieved by two basics operations: (1) removing the agent from its current host 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);
```

The 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 servers `s1` and `s2`. The following instructions express an agent migration from `s1` to `s2`:

```
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 what implies a migration. In fact, in a *proactive* mobile agent system, 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 makes it possible to express migration. But our system must furthermore support inter-agent communication to be useful.

5.3 Containers 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 the objects it contains. 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 that 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 which is storing `a2`, the key of `a2` and the method it wants to invoke[9].

5.4 The model of Active Containers

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 exist. An active container is then defined as:

$$\begin{array}{l} ActiveContainer \equiv (\lambda\ put). \\ \quad !\Big\{put(\lambda\ F, \widetilde{O}, handles).(F)(\widetilde{O}) \\ \qquad |(\nu\ get, call, remove) \\ \qquad\qquad \overline{handles}[get, call, remove]. \\ \qquad\quad \Big[!\overline{get}[F, \widetilde{O}] \\ \qquad\quad |\ !(call(\lambda\ m, args, result). \\ \qquad\qquad (\nu\ future)\ \overline{result}[future]. \\ \qquad\qquad\quad (\nu\ res)(\overline{m}[args, res] \\ \qquad\qquad\qquad\quad |\ res(\lambda\ val).\overline{future}[val]))\Big] \\ \qquad + remove \\ \quad \Big\} \end{array}$$

Even if problems related to the destruction of objects (*cf.* section 4.3) do not appear directly in this model, they are still here: after the reception of an empty message on the *remove* name, the services *get* and *remove* become unavailable. But, a previous invocation of *call* may have created many processes 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 [20].

Note that once an object has been inserted with the `put()` method, the name *handles* is used to further invoke the *get*, *remove* and *call* methods 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:

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

Considering the set:

$$\widetilde{A} \asymp \{onCreation, C, onMigration, M_o, onMigrating, M_i, onDisposing, D, run, R\}$$

Our agent server may be written:

$$\begin{array}{l} Server \equiv (\lambda\ \widetilde{S}) \\ \quad (\nu\ \widetilde{AC})(ActiveContainer)(\widetilde{AC}) \\ \qquad |(Services)(\widetilde{S}, \widetilde{AC}) \end{array}$$

$$\begin{array}{l} Services \equiv (\lambda\ \widetilde{S}, \widetilde{AC})!\ \Big\{ \\ \quad \widetilde{S}\#createAgent(\lambda\ \widetilde{A}, arg, returnid). \\ \qquad (\nu\ handles)(\nu\ id)\widetilde{AC}\#\overline{put}[Agent, \widetilde{A}, handles]. \\ \qquad handles(\lambda\ get, call, remove). \\ \qquad (\nu\ result)\overline{call}[\widetilde{A}\#onCreation, \widetilde{S}, arg, result]. \\ \qquad result(\lambda\ future).future(\lambda\ val). \\ \qquad (\nu\ dummy)\overline{call}[\widetilde{A}\#run, \mathbf{0}, dummy]\ | \\ \qquad \overline{returnid}[id]\ | \\ \qquad !\overline{id}[get, call, remove] \\ + \\ \quad \widetilde{S}\#sendAgent(\lambda\ id, \widetilde{S'}, returnNewid). \\ \qquad id(\lambda\ get, call, remove). \\ \qquad (\nu\ result)\overline{call}[\widetilde{A}\#onMigrating, \widetilde{S'}, result]. \\ \qquad result(\lambda\ future).future(\lambda\ val). \\ \qquad get(\lambda\ Agent, \widetilde{A}). \\ \qquad \overline{remove}. \\ \qquad \widetilde{S}\#receiveAgent(\widetilde{A}, returnNewid) \\ + \end{array}$$

[9] Method invocation is traditionally abstracted to message passing.

$$
\begin{array}{l}
\widetilde{S}\#receiveAgent(\lambda\ \widetilde{A}, returnid).\\
\quad (\nu\ handles)(\nu\ id)\widetilde{AC}\#\overline{put}[Agent,\\
\qquad\qquad\qquad\qquad \widetilde{A}, handles].\\
\qquad handles(\lambda\ get, call, remove).\\
\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onMigration,\\
\qquad\qquad\qquad\qquad \widetilde{S}, result].\\
\qquad result(\lambda\ future).future(\lambda\ val).\\
\qquad (\nu\ dummy)\overline{call}[\widetilde{A}\#run, \mathbf{0}, dummy]\ |\\
\qquad \overline{returnid}[id]\ |\\
\qquad !\overline{id}[get, call, remove]\\
+\\
\quad \widetilde{S}\#killAgent(\lambda\ id).\\
\qquad id(\lambda\ get, call, remove).\\
\qquad (\nu\ result)\overline{call}[\widetilde{A}\#onDisposing, \widetilde{S}, result].\\
\qquad result(\lambda\ future).future(\lambda\ val).\\
\qquad \overline{remove}
\end{array}
\Big\}
$$

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 synchronous 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 $\widetilde{A}$, the agent identifier makes it possible to retrieve the names *call*, *get* and *remove*. Sending an agent is achieved 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:

$$
\begin{array}{l}
Agent \equiv (\lambda\ id, \widetilde{A}).\\
(1)\quad (\nu\ getCurrentServer)\ \Big\{\\
(2)\quad (\nu\ setCurrentServer)\Big[\\
(3)\qquad [\ setCurrentServer(\lambda\ \widetilde{S}).\\
(4)\qquad\qquad \overline{getCurrentServer}[\widetilde{S}]\ ]\ |\\
\quad \Big[\widetilde{A}\#onCreation(\lambda\ \widetilde{S}, arg).\\
\qquad\quad \overline{setCurrentServer}[\widetilde{S}].(\widetilde{A}\#C)(arg)\ +\\
\quad \widetilde{A}\#onMigration(\lambda\ \overline{S}).\\
\qquad\quad \overline{setCurrentServer}[\widetilde{S}].(\widetilde{A}\#M_o)(\widetilde{S})\Big]\ \Big]\ |\\
\\
\quad \widetilde{A}\#run.(\widetilde{A}\#R)\ |\\
\\
\quad \Big[\widetilde{A}\#onMigrating(\lambda\ \widetilde{S}).(\widetilde{A}\#M_i)(\widetilde{S})\ +\\
\quad \widetilde{A}\#onDisposing.\\
\qquad\quad \overline{setCurrentServer}[\mathbf{0}].(\widetilde{A}\#D)\ \Big]\ |\\
\\
\quad \Big[\widetilde{A}\#migrate(\lambda\ \widetilde{S}').getCurrentServer(\lambda\ \widetilde{S}).\\
\qquad\quad \widetilde{S}\#\overline{send}[id, \widetilde{S}']\ +\\
\quad \widetilde{A}\#dispose.\\
\qquad\quad getCurrentServer(\lambda\ \widetilde{S}).\widetilde{S}\#\overline{kill}[id]\ \Big]\\
\Big\}
\end{array}
$$

Lines (1) to (4) model a single access variable. The methods *setCurrentServer* and *getCurrentServer* are self explanatory. The latter makes it possible for an agent to request its own migration *via* its *migrate* method.

Intuitively, it seems possible to simulate 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 raised in section 4.2. The work of Tom Melham [21] may ease the work using the proof system HOL [22].

6 Future Work 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 top of which many other things can be built [23, 20, 24]. It seems to be a good abstraction for distributed and/or parallel programming. As such, we propose the Mandala framework [25] which contains the JACOb [26] 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 an application called DJFractal [27].

Furthermore, observing the current situation, many actors in the community have changed their focus from mobile agent systems to distributed execution frameworks[10]. Furthermore, 54% of the links referenced on the Mobile Agent List [1] are invalid. One of the reasons [2] may be the fact, given in this article, that mobile agent systems are actually based on the active container abstraction. This abstraction eases 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.

[10] Following 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] P. 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, november, 8th 2004. `http://mandala.sf.net/docs/thesis.pdf`.

[3] C. Shen and L.T. Chen., 1998. "UCI Undergraduate Research Journal".

[4] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[5] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analysing mobile code languages. In *Second International ECOOP Workshop on Mobile Object Systems*, Linz, Austria, July 1996.

[6] H. Damir, C. Dragana, M. Veljko, K. Petar, and K. 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.

[7] IBM Corporation. Aglets home page, January 2001. `http://www.trl.ibm.co.jp/aglets/` et aussi `http://aglets.sourceforge.net/`.

[8] Recursion Software (purchased from ObjectSpace). Voyager home page, Octobre 2003. `http://www.recursionsw.com/products/-voyager/`.

[9] Object Management Group. Grasshopper home page, octobre 2003. `http://www.grasshopper.de/`.

[10] N. Kothari. AgentOS - A Java based mobile agent system. ICS Honors Project Final Report.

[11] A. Carzaniga, G.P. Picco, and G. Vigna. Designing distributed applications with a mobile code paradigm. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, USA, 1997.

[12] U. Nestmann. Links on calculi for mobile processes, January 2001. `http://lampwww.epfl.ch/mobility/`.

[13] R. Milner. The polyadic π-calculus : a tutorial. Technical report, Laboratory for Foundation of Computer Science, Computer Science Department, Edinburgh University, octobre 1991.

[14] D. Walker, R. Milner, and J. 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.

[15] R. Milner. *Communicating and Mobile Systems – The Pi Calculus.* Cambridge University Press, June 1999. ISBN:0521658691.

[16] Lange B. Danny and Oshima Mitsuru. *Programming and Deploying Mobile Agents with Java*, chapter Mobile Agents With Java: The Aglets API. 1998.

[17] Benjamin Pierce and Davide Sangiorgi. Behavioral equivalence in the polymorphic pi-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.

[18] Sun microsystem. Why are `Thread.stop()`, `Thread.suspend()`, `Thread.resume()` and `Runtime.runFinalizersOnExit()` deprecated? `http://java.sun.com/products/jdk/-1.2/docs/guide/misc/threadPrimitive-Deprecation.html`.

[19] C.G. Harrison, D.M. Chess, and A. Kershenbaum. Mobile Agents: Are they a good idea? Technical report, T. J. Watson Research Center, Yorktown Heights, New York, 1995.

[20] S. Chaumette and P. Vignéras. Extensible and customizable just-in-time security (JITS) management of client-server communication in java. In Joubert et al. [28], pages 321–327.

[21] T.F. Melham. A mechanized theory of the *pi*-calculus in HOL. Technical report, Departement d'informatique 'a l' universit'e de Glasgow, Ecosse, 1992.

[22] T.F. Melham. *Introduction to the HOL theorem prover.* University of Cambridge, Computer Laboratory, Cambridge, England, 1990.

[23] S. Chaumette and P. Vignéras. A framework for seamlesly making object oriented applications distributed. In Joubert et al. [28], pages 305–312.

[24] P. 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.

[25] P. Vignéras. Mandala. Web page, August 2004. `http://mandala.sf.net/`.

[26] S. Chaumette and P. Vignéras. Active containers: an alternative approach to mobile agents systems. Second International Symposium on Object Oriented Parallel Environments, ISCOPE 98., december 1998. Santa Fe, NM, USA. Poster.

[27] P. Vignéras. DJFractal. Web page, August 2004. `http://djfractal.sf.net/`.

[28] G.R. Joubert, W.E. Nagel, F.J. Peters, and W.V. Walter, editors. *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003, Dresden, Germany*, volume 13 of *Advances in Parallel Computing*. Elsevier, 2004.