Agent Mobility and Reification of Computational State

Werner Van Belle, Theo D'Hondt
Programming Technology Lab
Departement Informatica
Vrije Universiteit Brussel

{werner.van.belle|tjdhondt}@vub.ac.be

20 March 2000

Abstract

This paper describes an experiment with mobility in multi-agent systems. The setting is a virtual machine that supports reification of the computational state of a running process. The objective is to investigate how this feature facilitates telescripting and to speculate on how languages like Java should evolve to include the resulting notion of strong migration.

Mobile Agent Systems

An agent is a persistent and autonomous software component. It can be thought of as a process executed by some virtual machine but it is mainly intended to provide a particular service. Viewed as processes, agents run concurrently on one or more machines and have their own data space and computational state.

We talk about multi agents if two or more agents, represented by processes, are able to communicate with each other. Depending on the agent system this communication can be performed by means of the available technology (*Remote Procedure Call*, *Remote Method Invocation*, etc.). In our particular setup, we will require our communication mechanism to support asynchronous remote message passing.

A mobile agent is an agent which is able to move between different machines, or in a more abstract sense, between *locations*. We will discuss migration in the following section.

A mobile multi agent system is a software artefact specified in a language that provides sufficient expressiveness and flexibility to allow the construction of multiple interacting mobile agents. This includes routing of messages between agent systems and providing an interconnection with

other systems. The agent system has to take care of migration of agents, serialization of messages and scheduling of processes. This should, of course, have a minimum of direct impact on the agents themselves.
This definition of mobile multi agents differs from others which view an agent as an intelligent entity that interacts with some user on a creative basis: it should learn about its environment and adapt to it as needed. In this paper we are more interested in the architectural support, so we will focus on the design and implementation of mobile multi agent systems. Particular attention is paid to the migration aspect.

Migration

In this paper, migration denotes the act of transferring a running agent to another location. After migration, the agent should continue and proceed seamlessly with what it was performing before its move. In order to obtain this result, three actions should be undertaken to prepare, guide and complete the actual migration. First we have to encapsulate the agent's complete state; next, we need to transfer this capsule and finally, we need to restore and re-activate the agent in its new environment. With the wide availability of standard communication networks we can safely say that transfer itself is no longer a problem. The challenging aspect in migration is the wrapping and unwrapping of the agent in order to restore it to its full powers.

An encapsulated agent should not carry the complete agent system in its wrapping. We therefore need to make an inventory of those features that determine the working of an agent. Conceptually, an agent consists of

- A data section which contains the environment needed by the agent;
- A (shared) code section which describes the behavior of the agent
- One or more runtime structures which describe the computational state of the processes
- Connections to resources and to the underlying agent system

Today, almost no mobile multi agent system (except [1] and [2]), encapsulates and transfers the computational state of the agent's process. Typically, the agent program is duplicated on the receiving location and after migration the agent is reinitialized from data saved before the move. This is because it is a major challenge to the builder of a virtual machine to provide a mechanism for capturing the computational state, as will be shown later.

This difficulty led to two kinds of migration models.

The first kind, called the *looping model* [3], transfers all of the agent's constituent parts except for the computational state. This means that the

only way to encapsulate an agent correctly is to ensure that the computational state is empty. Consequently, the agent has to stop voluntarily prior to migration and afterwards start up again. As such this model *loops* between sequences of computation that start up and stop completely.
In our view this approach to migration is flawed. First of all, the looping model has a nefarious impact on how the agent's program is engineered, making the program code difficult to manage. Migration therefore becomes too difficult to use and as such will be avoided unless absolutely needed. Second, the agent itself is sole master of the migration process; it is impossible for some external agency to capture the agent's full state and direct migration from the outside. This prohibits the use of manager agents which send out agents to other machines as needed.

The second model of migration is called *telescripting* [1], also called strong migration. When an agent systems supports telescripting it allows the agent to move at all times to other locations, without the need of restarting the entire computation. As can be inferred, in this model the agent's computational state is transferred correctly.

Scripting an Agent

Before starting out on the fundamentals of building a virtual machine that supports reification of computation, we shall describe the experimental setting of this report.

In our approach we have started from an existing virtual machine called Pico [7] which features open semantics. Pico is accessible via an extremely simple language yet its expressiveness is very high, comparable to e.g. Scheme. Pico semantics are defined by a set of nine evaluation functions that are supported by a storage model and a computational model. The storage model features full storage management and reclamation; the computational model is based on a pushdown automaton that manages expressions and continuations on a double stack. Continuations, inspired by *continuation passing style* [4], are thunks that are sequenced in order to support computation. Pico requires less than twenty continuations to implement the complete semantics of the language.

In order to support our experiment, Pico semantics were extended to support objects and multi threading. The result, called *Borg* is a prototype-based language. Objects can specified and cloned in a very simple but effective way. In the transcript below `makecircle` denotes a mixin method which extends the basic `point` to become a `circle`:

```
createpoint(x,y)::
  { getx():: x;
    gety():: y;
```

```
    setx(nx):: x:=nx;
    sety(ny):: y:=ny;
    makecircle(r)::
            { setr(nr):: r:=nr;
              getr():: r;
              clone() };
    clone() }
:<closure createpoint>
a:createpoint(1,2)
:<dictionary>
b:a.makecircle(900)
:<dictionary>
b.getx()
:1
b.setx(8)
:8
b.getx()
:8
a.getx()
:1
```

Another abstraction layer we needed to support our experiment is routing of messages and naming of agents. The problem with existing distributed systems is that whenever an object changes its place (insofar as possible) its name changes to reflect its new position. *Borg* has an original naming service built into it so that we have a location transparant naming scheme and a hierarchical interconnection network. [5]

Uniform Message Sending

On top of this all we have installed a serializer to store and retrieve subgraphs of the data store. The serializer differentiates between two kinds of serialization. The first concerns expressions handled as messages sent to another location. The second concerns expressions handled as complete agents which should be sent to a remote host.

When a message is serialized, we traverse the data graph and store everything that we encounter on a stream. This process stops at leaves and at dictionaries which will be serialized as references to remote dictionaries. In this way, an agent can send a local dictionary to another agent without having to send the entire dictionary content to the communication partner.

Uniform Message Sending: We use this messages serializer as a means for sending messages to other agents in the same way we would send a message to an object. There is only one difference: the caller will not receive a return value and the execution will continue immediately. The expressions given as arguments to a remote function will be serialized and deserialized automaticaly. If there are too many expressions for an agent to evaluate, the agent will store them in a queue and evaluate them one by one.

Below is an example of two *Borg* programs that communicate with each other. The first program is the receiving agent which will do a callback to the second agent. After having installed a callback procedure the second agent calls the first one.

To all intents and purposes we have introduced asysnchronous message passing, very similar to an actor system [8].

<u>Agent Tecra/ses1</u>

```
Calculate(…,callback)::
  { <some calculation>;
    callback.Answeris(…) }
```

<u>Agent Tecra/ses2</u>

```
Answeris(…)::
  { display(“The answer is: “);
    display(result) };
agent: remotedict(“Tecra/ses1”);
agent.Calculate(…,agentself())}
```

We can see that this way of working has a number of advantages over standard distributed systems:

- We don’t have to generate and compile stubs beforehand. (in comparison to Java and CORBA this is definitely a strong advantage, particularly when we transfer code to other locations)
- The code is interpreted which guarantees small and powerful pieces of code (compiled code is much larger than interpreted code)
- We don’t have to take location of an agent into account, since an agent’s name doesn’t change after migration.
- Sending messages between agents is similar to sending messages between objects, with automatic serialization of messages.

Virtual Machine

A standard virtual machine for a simple language such as *Borg* consists of:

➢ A language processor which, given a program text, generates an instance of some abstract grammar (called expression). In *Borg* for example, the program

```
setr(nr):: r:=nr
```

will be translated into the following expression:

```
[DCL [APL setr [TAB [REF nr]]][ASS r [REF nr]]]
```

Expressions can be externally stored as a sophisticated kind of *bytecode* or they can be kept as executable code in some datastructure. *Borg* is very close to for instance *Scheme*, in that language and execution model practically coincide.

- A dictionary (or symboltable) to store identifiers and their values. The values are either inline, or are references to a subset of possible expressions (actually, values are expressions that have identity for the evaluation function). A *Borg* dictionary is effectively a stack of frames so as to support the notion of *scope*.
- An evaluation loop that unites the evaluation functions that make up *Borg* semantics; this loop accepts a sequence of expressions and evaluates them.

In the more general case, the complete process will take program text, convert it into an expression which will be dispatched to a specialised interpreter. For instance:

```
apply(exp)::
 if(exp.operand = '-',
   { par: evaluate(exp.par(1));
     number(-par.value) },
   if(exp.operand = '+',
     { par1: evaluate(exp.par(1));
       par2: evaluate(exp.par(2));
                  number(par1.value+par2.value) },
   error()))
```

then, for example an evaluation of

```
apply(-,apply(+,1,2))
```

will result in a call sequence of

```
apply(apply(-,apply(+,1,2)))
evaluate(apply(+,1,2))
evaluate(1)
evaluate(2)
```

The expressions used during this computation are typically stored on a stack, which in general coincides with the run-time stack of the program that implements the interpreter

Reification of the runtime stack

We will now describe a very straightforward way to reify the runtime stack of the agent interpreter, thus making the agent runtime stack a first order entity in the interpreted language. We will not describe full reification because it not needed for our application (strong migration of code), but we nevertheless need some way to capture the runtime stack and to handle it as if it were yet another object in the interpreted language and not only in the language the interpreter is written in. We view the computational model as a paired expression/continuation stack. Below are the rules which should be kept in thought when changing an existing interpreter.

- Whenever the interpreter is required to evaluate an expression it should not call other functions to divide the given expression and conquer a result.
- To retrieve the parameters passed to a function it has to take a peek at the expression stack and pop these parameters by itself.
- When another function has to be called (e.g. a function such as evaluate) it has to store the function to be called and the expression accompanying the function on the continuation stack.
- At the end of a given function it has to proceed with the next function on the continuation stack. In a tail recursive language this can be done by applying the result of contstack.pop() at the end of the function. If we don't have a tailrecursive language we need to return control and hope there will be an outerstackloop available.

The above interpreter code will be converted to

```
Minus()::
  { par: expstack.pop();
    expstack.push(number(-par.value)) }

Apply()::
  { exp: expstack.pop();
    if(exp.operand = '-',
       { contstack.push(minus);
         contstack.push(evaluate);
         expstack.push(exp.par(1)) },
       if(exp.operand = '+',
          { par1: exp.par(1);
            par2: exp.par(2);
            contstack.push(addfinal);
            contstack.push(addaux);
            expstack.push(par2);
            contstack.push(evaluate);
            expstack.push(par1) },
          error())) }

Addaux()::
  { result1: expstack.pop();
    exp2: expstack.pop();
    expstack.push(result1);
    expstack.push(exp2);
    contstack.push(evaluate) }

Addfinal()::
  { result1: expstack.pop();
```

```
result2: expstack.pop();
expstack.push(number(result1.value+result2.value)) }
```

Implications of this conversion are obvious:

- We can easily garbage collect the system, because we can put the running stacks in some root-table.
- We have an extra indirection which could lead to a suboptimal performance.
- We can easily implement stack optimizations because we have the stack under control. For example, implementing tail recursion is no problem if we change the apply somewhat: An application consists of changing the current dictionary, evaluating an expression and returning a value. If we check that there is already a return-result continuation we won't have to push a new one.

Migration revisited

An interpreter written in this way, i.e.with (a) the capability of serializing the data store and (b) the ability to reify the computational state, can easily be used to implement strong migration. Even if an agent process is in the midst of being evaluated, we simply interrupt the process, serialize the state of the entire computation, which also includes its computational state, and send it to another location while removing the proces from interpreter control. At the receiving end we deserialize the agent and start a process which uses the freshly deserialized computational state.

The above steps should be taken at a moment at which the stack is consistent, neither while a continuation thunk is executing, nor when there are global variables which are not saved in the data store. With only these things to think of we can state that we have implemented strong migration without too much difficulty.

Serializing agents consists of traversing the data graph and storing everything we encounter, including local dictionaries. The one exception to this is the root environment which is simply marked as being the root environment. In this way we can integrate agents in their new agent environment upon their arrival at a new location.

The example below illustrates how migration can be exploited. We see that we can call the agentmove native function at any time. When the agent arrives at its new location it will print ‘after moving’ without the need for restarting it at its main entry point.

```
Calcul(pars)::
  { <some calculation>;
   display(“before moving”);
   agentmove(remotedict(“otherplace.tecra”));
```

```
    display("after moving") }

  main(pars,callback)::
    callback.Answeris(calcul(pars))
```

Java

Java is the current state of the art programming language for distributed systems. At the moment of writing this paper, Sun is developping *Jumping Beans* [6]. These are based upon *Java Beans*, which define the notion of *software components* as well-defined interfaces which are contracts to be honored to ensure interoperability.
Jumping Beans require a number of properties to be met by both an application and its host environment for this application to become mobile:

- The application should be serializable.
- It must retrieve resources in a location-independent fashion If one wants to use techniques which are application specific, one must be aware that the host environment can change. A better way to retrieve resources is to write applications in a mobility-aware fashion by techniques which are specific to *Jumping Beans*.
- If the mobile application requires *Jumping Beans* services or requires notification of Jumping Beans events, then it must implement MobileApp.

Furthermore the *bean* has to implement a number of hooks typically found in weak migration schemes. These hooks are present after creation, before dispatching, after arrival, before deactivation, after reactivation and before destruction.

In order to make *Jumping Beans* (or similar migration schemes) really workable**,** it may become necessary to rework the Java virtual machine so that one can serialize the computational state of a thread as if it were yet another standard Java object. Without this reification of the computational state, mobile applications are doomed to remain complex artefacts, well outside of the mainstream of software engineering.

Conclusion

In this paper we have reported on an experiment in mobility of software agents. In particular, we investigated the reification of the computational state of an agent's underlying process, as a basis for the actual migration scheme. We extended a simple, experimental virtual machine with first-class computations and we proceeded by using these to transmit an active agent from one location to another without direct impact on the agent's specification. We conclude that this is a workable approach to bring the notion of mobility into the mainstream of software development. The next step should be to rework the results of this experiment into recommendations for future extensions to production-oriented environments such as Java.

References

[1] Jim White, *Mobile Agents White Paper*, General Magic

[2] Robert S. Gray, *Agent TCL: A flexible and secure mobile-agent system,* Department of Computer Science, Dartmouth College

[3] Gian Pietro Picco, *Understanding Code Mobility*, Politecnico di Torino, Italy, Tutorial at ECOOP98, 22 July 1998

[4] Gerald Jay Sussman, Guy Lewis Steele Jr, *Scheme, An Interpreter for Extended Lambda Calculus,* Massachusetts Institute of Technology, Artificial Intelligence Laboratory. December 1975

[5] Werner Van Belle, *Reinforcement Learning as a Routing Technique for Mobile Multi Agent Systems*, April 1997

[6] Ad Astra, *Jumping Beans*, *a white paper* , 1 september 1998

[7] Online documentation on http://pico.vub.ac.be

[8] Agha G. *Actors: A Model of Concurrent Computation in Distributed Systems,* AI Tech Report 844, MIT, 1985