Scalable Computing: Practice and Experience
Volume 11, Number 2, pp. 149–159. http://www.scpe.org
ISSN 1895-1767

MANAGEMENT OF HIGH PERFORMANCE SCIENTIFIC APPLICATIONS USING MOBILE AGENTS BASED SERVICES

SALVATORE VENTICINQUE*, ROCCO AVERSA*, BENIAMINO DI MARTINO*, RENATO DONINI*, SERGIO BRIGUGLIO†, AND GREGORIO VLAD†

Abstract. High performance scientific applications are currently implemented using native languages for optimizing performance and utilization of resources. It often deals with thousands of code lines in FORTRAN or in C built of legacy applications. On the other hand new technologies can be exploited to improve flexibility and portability of services on heterogeneous and distributed platforms. We propose here an approach that allows programmers to extend their applications to exploit this kind of services for management purposes. It can be done simply by adding some methods to the original code, which specialize application management on occurrence of particular events. We mean that applications do not need to be rewritten into different languages or adopting specific programming models. We implemented a native console that is used by Mobile Agents to control the application life-cycle. Agents implement a mobile service that supports check-pointing, suspension, resume, cloning and migration of managed applications. A WSRF interface has been provided to Grid users who do not need to be aware about agents technology. We used a FORTRAN code for simulation of plasma turbolence as a real case study.

1. Introduction. We aim here at investigating how Mobile Agents technology can be used to develop advanced services for management of resources in distributed systems. Mobile Agents mechanisms such as autonomy, reactivity, clone-ability and mobility can be exploited for resource management and load balancing when system conditions change dynamically. Most of all mobile agents platforms are executed by Virtual Machines which make transparent the hardware/software architecture of the hosting node. It allows to distribute and execute mobile agents code on heterogeneous environments in a flexible way. On the other hand, most of legacy applications have been implemented by languages such as FORTRAN and C, and they are compiled for a target machine in order to optimize their performances. Here we present a mobile agent based service that allows for management of native applications on heterogeneous distributed systems. Agent technology has been used to provide management facility on any node where the execution of applications will be started. Programmers, in order to exploit the management service, can extend their application without modify the original code, but by overriding some methods which specialize the application life-cycle. We aim at supporting checkpoint, resume, migration and monitoring. Furthermore service is targeted to each Grid user, who is unaware about Agent technology and can exploit services facilities by a compliant WSRF interface. A console that allows the application control by an agent has been designed and implemented. An agent based service designed and implemented to automatically perform management strategies. In the second section related works on management services and load balancing mechanism are described. In section 3 we describe the facilities we have implemented in order to control the application life-cycle. In section 4 the software architecture of our service is presented. Section 5 provides an example of simple application that has been extended in order to exploit the service.

2. Related work. Application management and migration are mechanisms developed in many environments for common purposes. There are many platforms which exploit migration to implement load balancing in distributed and parallel systems. When we deal with homogeneous clusters, process migration is supported to share resources dynamically and adaptively. MOSIX [1] provides a set of algorithms to react in real time to the changes of resources utilization in a cluster of workstations. Migration of processes is preemptive and transparent. The objective is to provide high performances to parallel and sequential applications. OpenMosix [2] is a decentralized version of MOSIX. Each node behaves as an autonomous system. Kerrighed [3] supports thread migration. When a node is less loaded a process is moved there from a more busy node. OpenSSI [3] does not need to save part of the process on the original hosting node. System calls are called locally. Not all the processes are candidates for migration.

On the other hand in heterogeneous systems process migration is not generally supported. Different environments are virtualized by a middleware that hides architectural details to the applications and supports portability. In this case is possible to migrate code and status of applications, but not to resume the process status. An hybrid approach that manage a Grid of homogeneous clusters is presented in [4] as an advance

*Second University of Naples, Aversa (CE), Italy, emails: {rocco.aversa, salvatore.venticinque}@unina2.it, renato.donini@gmail.com, beniamino.dimartino@unina.it
†Associazione EURATOM-ENEA sulla Fusione, Frascati, Rome, Italy,emails: {brigulgio, vlad}@enea.it

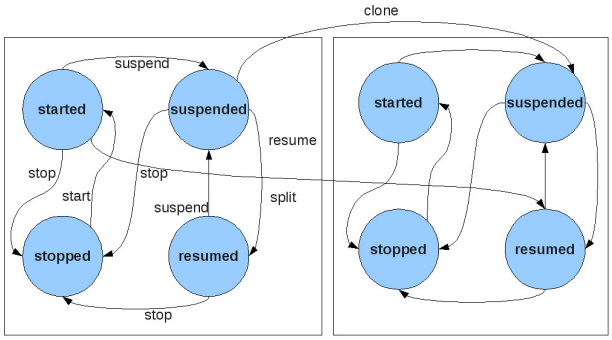

FIG. 3.1. *Application life-cycle*

of research cited above. Some relevant contributions, which exploit Mobile Agents technology are [5, 6]. We aim at exploiting flexibility of Mobile Agent programming to manage legacy applications without changing the original code and without rewriting the application into another language or adopting a new programming paradigm. Some approaches which should be compared with the one presented in this paper are cited below. [7] presents an approach to support mobility of applications in a GRID environment. A mobile agent is described by an UML-like language called blueprint. The blueprint description is transferred together its status. The receiving platform translates the agent blueprint description in a Java or a Python application that implement the original functionalities Blueprint is a language for a high level specification of agent functionalities and of its operational semantics. A specification describes the behavior of an agent in terms of its basic building blocks: components, control flow and data flow. Agent factories interpret the agent blueprint description and generate executable code composed of, for example, Java, Python, or C components. To support migration of common user applications, in [8], authors provide the possibility to insert some statements, such as go( ) or hop( ), between blocks of computation, such as for/while loops. A pre-compiler, such as ANTLR for C/C++ and JavaCC for Java source code, is used to substitute these statements with code that implements mobility. A user program may be entirely coded in C/C++ and executed in native mode. As a result Java agent starts the native code using a wrapper implemented by the JNI technology. In our approach programmers who want to support migration do not need to deal with any models, but they have just to handle such events which ask to save or resume the application status.

3. Management facilities and application life-cycle. We exploited Mobile Agents technology to allow services execution on heterogeneous platforms. We mean that we can execute monitoring and control facilities on heterogeneous nodes in a distributed environment. Furthermore we can move dynamically a service instance from a node to another resuming the execution at destination. Mechanisms of Mobile Agents technology have been adopted to design and implement advanced management facilities. Besides we provide the possibility to extend the same facilities to the managed application. Our model of application life-cycle is shown in Fig: 3.1. Regardless of the location, the process state could assume the following values: started, suspended, stopped and resumed. In the suspended mode, the application status has been saved in order to allow process restoring when a resume action will be invoked. Let us clClrify that the application status is saved not the process one. That's because we aim at supporting mobility across heterogeneous architectures. Life-cycle can be monitored and controlled by a software console that generates and handles the following events:

1. *start*: starts the execution of a native application on a cluster node
2. *suspend*: suspends the native application saving its execution status.
3. *resume*: resumes the native application restoring the status saved on the last suspension.
4. *stop*: stops the native application.
5. *checkpoint*: saves the status of the native application without stopping its execution.
6. *migrate*: migrates the native application to a target node restoring its status at destination.
7. *split*: clones the native application and splits the job between the clones.

We have developed both an interactive GUI and a batch interpreter for submitting commands to the application console.

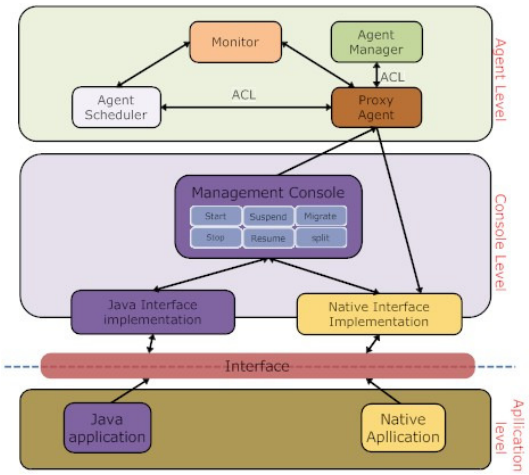

FIG. 4.1. *Software Architecture*

4. Service architecture. As shown in Fig:4.1 software architecture of the management service is composed of different elements at different levels. A first set of components implements a portable service executed by Mobile Agents. Distributed Agents perform different roles. They implement user interface, application monitoring and control, code management. A monitor detects relevant changes of system conditions and notifies these events to the Agent Scheduler. The Agent Scheduler reacts in order to avoid performance degradation. It communicate and coordinates other agents. The Proxy Agent is responsible of the application execution. It receives requests from other agents and manages the application by a Java console. An abstract console defines the interface between the proxy and the application. It is obviously independent by the kind of application that will be controlled. Native applications will be linked to the console by mean of a native implementation of abstract methods. In order to support the execution on heterogeneous architectures, the programmer has to make available a new shared library that overrides native methods and that has been compiled for a target architecture. Libraries will be stored on a remote repository and the proxy agent is able to automatically download them, when the execution is moved to a new heterogeneous node.

4.1. Agent level.

4.1.1. Agent Proxy. An Agent Proxy is delegated to manage an instance of users' application. It can:

- save and resume the status of the native application (save, resume);
- migrate, clone or split the application;
- handle special conditions such as termination.

For instance whenever the agent manager requires the migration of a native application to a selected node, the Agent Proxy has to transparently:

- suspend the native application as soon as possible;
- save the status of the native application;
- migrate to the target node;
- detect the target node hardware and software equipment;
- whenever it is necessary download and install the right library;
- restore the status of the native application;
- resume the native application.

4.1.2. Agent Manager. The Agent Manager provides a graphic interface for creating, migrating, cloning and destroying both Agent Proxies and the native applications. It sends standard ACL messages (ACL stands for Agent Communication Language) to ask for the actions ProxyAgents should take. The Manager allows to load references to libraries which have been built and made available for different hardware and software architecture. It can handle independently multiple execution instances of the same application. A snapshot of the Management Console GUI is shown in Figure 4.2.

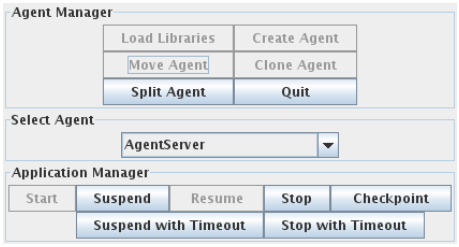

FIG. 4.2. *Management Console Gui*

4.1.3. Agent Monitor. Our architecture includes a monitor module to detect relevant changes of system conditions and notifies these events to the Autonomous Agent. As shown in the following the system is able to react to the events which affect the performance of the both service and application. The way to generate the events to send to the Batch Agent, could be less o more complex according to the particular requirements. Currently a simple configuration of management strategy based on threshold mechanisms is supported. The agent monitor checks the application's performance and compares it with a target input, such as the throughput of a web server; when a performance degrade was detected the right actions could be performed.

4.1.4. Agent Scheduler. The Agent Scheduler uses management facilities provided by Agent Proxies in order to carefully distribute the cluster workload or to optimize performance of applications. For instance, it can migrates native applications from a platform to another one that is less loaded. It can redistribute groups of applications in order to reduce network traffic by minimizing inter-communications, or reducing the overhead due to data transfer. Actually the user can configure the Scheduler behavior by associating a set of actions to a notification event. When a specific ACL messages has been received from the agent monitor the related batch file is interpreted and executed. A batch file can be written as described in Figure 4.1.4. In the example we show a sequence of commands which are executed on the occurrence of two events: *idle_node* and *busy_node*. Parameters of commands are extracted from content of related events notified by ACL messages.

```
<?xml version="1.0" encoding="UTF-8" ?>
<batch>
 <activation>
   <and>
     <event name="idle_node">
     <event name="busy_node">
   </and>
 </activation>
 <sequence>
  <operation>
   <command type="suspend" agent="$busy_node.agent_name[0]"/>
  </operation>
  <operation>
   <command type="move" agent="$busy_node.agent_name[0]">
    <parameters>
     <parameter>$idle_node.container[0] </parameter>
    </parameters>
   </command>
  </operation>
  <operation>
   <command type="resume" agent="$busy_node.agent_name[0]"/>
  </operation>
 <sequence>
</batch>
```

FIG. 4.3. *An XML batch file define the list of action that must be taken on an event occurence*

We are planning to adopt a more complete language such as the ones which support choreography or orchestration [9].

4.2. Console level. In order to make transparent the management of different kinds of applications we defined the ApplicationManager abstract class. An implementation needs to support management of each kind of application. We provided a NativeManager class with java and native methods. Other classes implements special utilities. In particular:

- ApplicationManager: is an abstract class that represents an application manager.
- NativeManager: is a class that extends ApplicationManger It overrides abstract methods in order to interface with native applications.
- ApplicationThread: is a thread that starts native application and waits for events.
- Library: is a class that contains a set of parameters, which are used to identify and retrieve the compliant version of application library for the target machine architecture.
- ManagementException: is a class that implements an exception that could be thrown by the ApplicationManager.

JNI (Java Native Interface) is the technology that has been used to link the native implementation of the ApplicationManager to the Java implementation. JNI defines a standard naming and calling convention so the Java virtual machine can locate and invoke native methods. A native implementation of these methods have been developed in POSIX C and compiled for different architectures (AMD64, IA32, ...). As it is shown in Figure 4.4, in order to allow its application to be managed a programmer has to add those methods which override the ones defined in the native console. Linking the original application code with the new methods and the native console, a dynamic library for a target machine can be built. The native console is implemented by two POSIX processes: a father and its son. The son is spawn when the application starts. The father waits for requests from the Java part of the service, forwards them to the son that executes the application code by POSIX signals. The son before to start registers signal handlers and communicate by two POSIX pipes with the father. Pipes are used by the son to communicate to the father the file name where the application status has been stored and to communicate events such as a termination. Handlers can be specialized by the programmer by overriding the following methods (using C, FORTRAN, or any other native languages): *start_()*, *resume_(char status[])*, *stop_(char status[])*, *suspend_(char status[])*, *checkpoint_(char status[])*, *split_(char status[])*, whose meanings has been described in the previous sections.

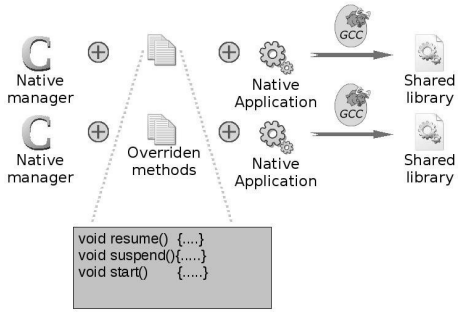

FIG. 4.4. *Building a new library*

5. Plasma turbulence simulation. As case study, we chose a Fortran application for Particle-in-cell simulation. Particle-in-cell simulation consists [15] in evolving the coordinates of a set of N_{part} particles in certain fluctuating fields computed (in terms of particle contributions) only at the points of a discrete spatial grid and then interpolated at each particle (continuous) position. Two main strategies have been developed for the workload decomposition related to porting PIC codes on parallel systems: the *particle decomposition* strategy [11] and the *domain decomposition* one [12, 13]. Domain decomposition consists in assigning different portions of the physical domain and the corresponding portions of the spatial grid to different processes, along with the particles that reside on them. Particle decomposition, instead, statically distributes the particle population among the processes, while assigning the whole domain (and the spatial grid) to each process. As a general fact, the particle decomposition is very efficient and yields a perfect load balancing, at the expenses of memory overheads. Conversely, the domain decomposition does not require a memory waste, while presenting

particle migration between different portions of the domain, which causes communication overheads and the need for dynamic load balancing [14, 13]. The typical structure of a PIC code for plasma particle simulation can be represented as follows. At each time step, the code

1. computes the electromagnetic fields only at the N_{cell} points of a discrete spatial grid (*field solver* phase);
2. interpolates the fields at the (continuous) particle positions in order to evolve particle phase-space coordinates (*particle pushing* phase);
3. collects particle contribution to the pressure field at the spatial-grid points to close the field equations (*pressure computation* phase).

We can schematically represent the structure of this time iteration by the following code excerpt:

```
call field_solver(pressure,field)
call pushing(field,x_part)
call compute_pressure(x_part,pressure)
```

Here, `pressure(1:n_cell)`, `field(1:n_cell)` and `x_part(1:n_part)` (with `n_cell`= N_{cell} and `n_part`= N_{part}) represent pressure, electromagnetic-field and particle-position arrays, respectively. In order to simplify the notation, we will refer, in the pseudo-code excerpts, to a one-dimensional case, while the real code refers to a three-dimensional (3-D) application. In implementing a parallel version of the code, according to the distributed-memory domain-decomposition strategy, different portions of the physical domain and of the corresponding spatial grid are assigned to the n_{node} different nodes, along with the particles that reside on them. This approach yields benefits and problems that are complementary to those yielded by the particle-decomposition one [11]: on the one hand, the memory resources required to each node are approximately reduced by the number of nodes (`n_part`$\sim N_{part}/n_{node}$, `n_cell`$\sim N_{cell}/n_{node}$); an almost linear scaling of the attainable physical-space resolution (i. e., the maximum size of the spatial grid) with the number of nodes is then obtained. On the other hand, inter-node communication is required to update the fields at the boundary between two different portions of the domain, as well as to transfer those particles that migrate from one domain portion to another. Such a particle migration possibly determines a severe load unbalancing of the different processes, then requiring a dynamic balancing, at the expenses of further computations and communications. Let us report here the schematic representation of the time iteration performed by each process, before giving some detail on the implementation of such procedures:

```
 call field_solver(pressure,field)
 call check_loads(i_check,n_part,n_part_left_v,
&    n_part_right_v)
 if(i_check.eq.1)then
  call load_balancing(n_part_left_v,n_part_right_v,
&     n_cell_left,n_cell_right,n_part_left,n_part_right)
  n_cell_new=n_cell+n_cell_left+n_cell_right
  if(n_cell_new.gt.n_cell)then
   allocate(field_aux(n_cell))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell)=field_aux(1:n_cell)
   deallocate(field_aux)
  endif
  n_cell=max(n_cell,n_cell_new)
  n_cell_old=n_cell
  call send_receive_cells(field,x_part,
&     n_cell_left,n_cell_right,n_part_left,n_part_right)
  if(n_cell_new.lt.n_cell_old)then
   allocate(field_aux(n_cell_old))
   field_aux=field
   deallocate(field)
   allocate(field(n_cell_new))
   field(1:n_cell_new)=field_aux(1:n_cell_new)
   deallocate(field_aux)
```

```
 endif
 n_cell=n_cell_new
 n_part=n_part+n_part_left+n_part_right
endif
call pushing(field,x_part)
call transfer_particles(x_part,n_part)
allocate(pressure(n_cell))
call compute_pressure(x_part,pressure)
call correct_pressure(pressure)
```

In order to avoid continuous reallocation of particle arrays (here represented by `x_part`) because of the particle migration from one subdomain to another, we overdimension (e.g., +20%) such arrays with respect to the initial optimal-balance size, N_{part}/n_{node}. Fluctuations of `n_part` around this optimal size are allowed within a certain band of oscillation (e.g., ±10%). This band is defined in such a way to prevent, under normal conditions, index overflows and, at the same time, to avoid excessive load unbalancing.

5.1. Restructuring the original code. Preliminary experiments deal with restructuring and management of the sequential program that solves the problem presented above. The program performs, after an inizialization phase, a number of iterations that at each run update values of fields, pressures, position and velocity of simulation particles. The original application (about 5-thousands lines of fortran code) has been provided by the ENEA[1]. It saves at the end of each iteration the intermediate results. We indentified the set of additional relevant information that need to be saved, at the end of each iteration in order to checkpoint and resume the application after a suspension. It means that before a new iteration starts, the application status has been already saved. A suspension will cause the lost of work that has performed since the beginning of the current iteration.

We restructured the original code by adding a number of functions to handle the *start*, *stop*, *suspend* and *resume* events. *start:* it calls the *init* function that reads the input data. The *init* subroutine starts also the main cycle that use many iterations to update particles data step by step.

```
integer function start()
     start=1
     call init
     call main_cycle
     end function
```

stop: it stops the execution, that will restart from the beginning of the application when the start-command will be invoked. When the application restarts input data will be read again and intermediate results will be lost.

```
integer function stop()
     stop=1
     end function
```

checkpoint: it writes in a file the intermediate information which describe the particles (pressure, position, volume ...), algorithm data such as the current iteration, the max number of iterations, and others. These data are stored in two modules, which are *global_data* and *particles_data*. The *write_data* subroutine uses the *dataout* parameter as filename.

```
integer function checkpoint(dataout,n)
          integer n
      character dataout(n)
          use global_data
          use particles_data
          checkpoint=1
          call write_data(dataout)
         end function
```

suspend: it suspends the execution that can be resumed at any time, restoring the application status. The write status saves the application status in a file as it has been explained above. If a suspension has been requested,

[1]ENEA is the Italian Agency for New Technologies, Energy and Sustainable Economic Development - *http* : *//www.fusione.enea.it*

Iteration	StartTime (sec)	Duration (sec)
1	5,864	86,414
2	92,278	87,236
3	179,514	87,574
4	267,087	87,836
5	354.923	87,657
6	442,580	87,835
7	530,414	87,792
8	618,206	87,695
9	705,900	87,658
10	793,558	87,620

FIG. 6.1. *Mean time for each iteration of plasma simulation*

the function is executed at the end of the current iteration. In this case it does not need to perform any actions.

```
integer function suspend()
        suspend=1
        end
```

*resume:*it resumes the application execution after a suspension. The *read_data* subroutine restores the application status. The *read_data* subroutine initializes the gobal varaiables of the program modules with the values stored in the *indata* file. The *start_main_cycle* starts from the beginning of the same iteration that was interrupted by the last suspension.

```
integer function resume(indata,n)
         integer n
         character indata(n)
             use global_data
             use particles_data
             resume=1
         call read_data(indata)
         call start_main_cycle
         end function
```

Migration is transparent to the programmer. It transfer the Java agent code and the *data_file*, nedded for resumption.

6. Experimental results. In order to build the native library we used the Ubuntu Linux Operative System, GNU FORTRAN 4.4.1, GCC 4.4.1 and SUN JDK 1.6. The Jade platform v. 1.3.5 has been used to develop and run agents. The *C* implementation of the native console and the FORTRAN restructured application have been compiled and linked together. Agents can download different builds, for 64 bit and 32 bit Linux platforms by a FTP server. Computing nodes are connected by a 100MB Ethernet network. Successful experiments demonstrated that the prototype works properly, in fact the platform successfully supports checkpointing, resume, migration and monitoring of native applications. Results allowed us to evaluate the overhead due to cloning and migration when it is necessary to migrate the native code which was previously compiled for the target machine, and the status of execution.

We experienced the native management of the application for the plasma simulation. The original code has been executed on a dual-cpu Intel Xeon 3.06 GHz, 512K cache size and 2GB RAM. In a first experiment an *Agent Proxy* is started from an *Agent Manager* on its same node. The *Agent Proxy* get the cpu architecture and the operative system (i386,linux) and asks for the correct version of native library. A shared library is downloaded via a public ftp server connected to the Internet by a 8MB connection in a different geographic site . The application executes 10 iterations without suspension and resume, but saving at each iteration the intermediate results of its computation in case of asynchronous requests for migration. In Table 6 we show the measures we have collected after 100 runs. Wen can see that time needed to start the application, and to read the input data is 5,86 sec. Before that the time needed to ask for the shared libraries, to get their location and download it was 1,794 (sec). In a second experiment the *Agent Manager* sends randomly a request for

migration from a node to another one of the cluster. Even if we know in advance that the nodes have the same cpu an operative system we chose to ask for and download again the shared library. Furthermore, even if the nodes share the same file system via NFS, however the agent servers and the application execute in two different directory. In the first one we have the agents code and the input data, in the second one only the intermediate results will be saved. The agent code will be transfered during migration and saved in a cache. The status of application contains intermediate values of the program variables and of the particles. The total size of information is about 95MB. To optimize the data transfer, the application compresses the data before the migration, and decompresses the file at destination. The time spent to suspend, migrate and resume the application will be:

$$T_{total} = T_{write} + T_{compress} + T_{decompress} + T_{transfer} + T_{resume} + T_{lost} \tag{6.1}$$

When migration is asked the array of particles has been already written into a file at the end of the previous iteration. T_{write} represents the time needed to write only the current values of variables necessary for resumption. $T_{compress}$ is the compression time that takes 6.2 sec, while the decompression takes 1.4 sec. The compression reduce the data size to 30 MB. In the *resume* method of the native console, overridden by the application, we call the tar utility, with the czf options to compress the application status. It could be done in the java code if such utility is not available. The *Proxy Agent*, before to move, store the compressed file in a byte array and uses the Jade APIs to migrate with its own code and attributes. $T_{transfer}$ is the time needed to transfer the agents and the application status. On the target node we only need a running agent platform. T_{resume} represents the time to restart the application that will read the intermediate values of its variable from the file system. T_{lost} is the time elapsed since the beginning of the current iteration before the migration request. It depends on when the signal interrupts the main thread. It will be less than the duration of a complete iteration. In our implementation all the work that has been done since the beginning of the current iteration is lost. In the formula we do not consider the time needed to download the native code because we could send an agent at the new destination to identify the target architecture and to download the right version of native library before and meanwhile the application is suspended. However in the following measures this strategy as not been applied and T_{total} will include also that contribution.

After a test-set of 100 runs, we observed a mean turnaround of 16m 22,611s without migration, and 17m 18,993s with a random migration. That means a mean time of T_{total} equals to 56,382 secs to be spent for migrating the execution from a node to another one of the local network. We can observe that the migration, in this case, takes about 64% of a single iteration, that is not relevant if compared to a real execution of hundreds iterations. We mean that the service can be effectively used to move the execution to a faster node, or to continue when the hosting machine should be upgraded or restarted. Furthermore many performance improvements can be designed. For example the choice to save the application status at each iteration is not the most effective one. We could plan different strategies taking into account the duration of each iteration, the time needed to suspend, migrate and resume the application, and the probability that it could be happening. In Fig. 6 it is shown the time trace for one execution of the test-set. We can see step by step what happened. In the first column we have *who takes the time*, in the second one the *item* is described and in the last one the time duration. On the left side we describe what happened on the first node (Container 1) and on the right side the second Container nodes works after the migration. The character @ represents the time elapsed since the start of the native application, otherwise the time value represents the duration of that event. In this case the suspension is received at 15,073 sec from the init of the application and T_{lost} is about 9,17 secs. As the mean iteration lasts for 88,728 secs, the other contributions of T_{total} are all relevant. We can observe that the library download takes more time at Container-2 because the Agent Proxy is running on a different node than the Agent Manager, so the communication use the cluster network. Nevertheless the application needs to resume the intermediate results before to start the first iteration, however all the needing initialization are already done. That is because the first iteration starts already at 0,35 (sec).

7. Conclusions. We presented an approach for agents based management of high performance scientific applications, providing a real case study as a proof of concept. We designed and implemented a Mobile Agents based service and its interface to POSIX applications. Programmers can extend their native applications in order to support checkpoint, migration, suspension, resuming, etc. Service implementation is in Java. It is portable and mobile. Application portability on heterogeneous node is supported through the dynamic linking of native libraries compiled for the target machine. The application life cycle can be controlled interactively from

Container-1			Container-2		
Code	Event	Time (sec)	Code	Event	Time
Proxy Agent	Library down.	1,629	- - - - -	- - - - -	- - - - -
Application	Iteration 1	@5,893 ; 0	- - - - -	- - - - -	- - - - -
Application	Suspend	@15,073	- - - - -	- - - - -	- - - - -
Agent manager	T_{total}	17,349	- - - - -	- - - - -	- - - - -
- - - - -	- - - - -	- - - - -	Proxy Agent	Library down.	2,015
- - - - -	- - - - -	- - - - -	Application	Iteration 1	@0,329 ; 89,168
- - - - -	- - - - -	- - - - -	Application	Iteration 2	@89,497 ; 89,194
- - - - -	- - - - -	- - - - -	Application	Iteration 3	@178,691 ; 89,186
- - - - -	- - - - -	- - - - -	Application	Iteration 4	@267,878 ; 89,047
- - - - -	- - - - -	- - - - -	Application	Iteration 5	@356,926 ; 88,348
- - - - -	- - - - -	- - - - -	Application	Iteration 6	@445,274 ; 88,615
- - - - -	- - - - -	- - - - -	Application	Iteration 7	@533,889 ; 88,537
- - - - -	- - - - -	- - - - -	Application	Iteration 8	@622,426 ; 88,364
- - - - -	- - - - -	- - - - -	Application	Iteration 9	@622,426 ; 88,88
- - - - -	- - - - -	- - - - -	Application	Iteration 10	@710,790 ; 87,932

FIG. 6.2. *Trace of a migrated execution*

a GUI or can be programmed by a scheduler that reacts to changes in the environment. An abstract console defines the methods which are used by agents in order to interface with the application. An implementation for POSIX application has been implemented and tested. We presented preliminary results which will be extended and discussed in the camera ready version. We Future work will deal with the management of parallel and distributed applications by cooperation of mobile agents which implement strategies for distributing the workload in order to optimize system utilization or application performance.

REFERENCES

[1] Barak A. and La'adan O., The MOSIX Multicomputer Operating System for High Performance Cluster Computing. Journal of Future Generation Computer Systems (13) 4-5, pp. 361-372, March 1998.
[2] J. Bilbao, G. Garate, A. Olozaga, A. del Portillo: Easy clustering with openMosix, World Scientific and Engineering Academy and Society (2005)
[3] R. Lottiaux, B. Boissinot, P. Gallard, G. Valle, C. Morin: openMosix, OpenSSI and Kerrighed: a comparative study, INRIA (2004)
[4] Maoz T., Barak A. and Amar L., Combining Virtual Machine Migration with Process Migration for HPC on Multi-Clusters and Grids, IEEE Cluster 2008, Tsukuba, Sept. 2008.
[5] I. Foster, N. Jennings, Kesselman, Brain Meets Brawn: Why Grid and Agents Need Each Other, in Proceeding of the 3rd Joint Conference Autonomous Agents and Multi-Agent Systems, pp. 8-15, 2004.
[6] B. Di Martino, O. F. Rana, Grid Performance and Resource Management using Mobile Agents, in: V. Getov et al. (Eds.), Performance Analysis and Grid Computing, pp. 251-264, Kluwer Academic Publishers, 2004 (ISBN 1-4020-7693-2).
[7] F. M. T. Brazier, B. J. Overeinder, M. van Steen, and N. J. E. Wijngaards, "Agent Factory: Generative Migration of Mobile Agents in Heterogeneous Environments", Proceedings of the 2002 ACM symposium on Applied computing, ISBN:1-58113-445-2, pp 101 - 106.
[8] M. Fukuda, Y. Tanaka, N. Suzuki, L. F. Bic, S. Kobayashi, "A mobile-agent-based PC grid", Autonomic Computing Workshop, ISBN: 0-7695-1983-0, pp 142- 150, June 2005.
[9] C. Peltz. Web Services Orchestration and Choreography. IEEE Computer Magazines, 2003.
[10] R. Donini, R. Aversa, B. Di Martino e S. Venticinque. Load balancing of mobile agents based applications in grid systems. In Proceedings of 8 IEEE 17th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Rome, 23-25 June 2008. IEEE.
[11] Di Martino, B., Briguglio, S., Vlad, G., Sguazzero, P.: Parallel PIC Plasma Simulation through Particle Decomposition Techniques. Parallel Computing **27**, n. 3, (2001) 295–314.
[12] Fox, G. C., Johnson, M., Lyzenga, G., Otto, S., Salmon, J., Walker, D.: Solving Problems on Concurrent Processors (Prentice Hall, Englewood Cliffs, New Jersey, 1988).

[13] Ferraro, R. D., Liewer, P., Decyk, V. K.: Dynamic Load Balancing for a 2D Concurrent Plasma PIC Code, J. Comput. Phys. **109**, (1993) 329–341.
[14] Cybenko, G.: Dynamic Load Balancing for Distributed Memory Multiprocessors. J. Parallel and Distributed Comput., **7**, (1989) 279–391.
[15] Birdsall, C. K., Langdon, A. B.: Plasma Physics via Computer Simulation. (McGraw-Hill, New York, 1985).

Edited by: June 21, 2010
Received: March 30, 2010
Accepted: June 21, 2010