Mobile Information Systems 4 (2008) 131–146
IOS Press


# A skeleton based programming paradigm for mobile multi-agents on distributed systems and its realization within the MAGDA Mobile Agents platform

R. Aversa[a], B. Di Martino[a], N. Mazzocca[b] and S. Venticinque[a]
[a]*Department of Information Engineering, Second University of Naples, Aversa, Italy*
*E-mail: {rocco.aversa,beniamino.dimartino,salvatore.venitcinque}@unina2.it*
[b]*Dipartimento di Informatica e Sistemistica, Universitá Federico II di Napoli, Napoli, Italy*
*E-mail: n.mazzocca@unina.it*

**Abstract.** Parallel programming effort can be reduced by using high level constructs such as algorithmic skeletons. Within the MAGDA toolset, supporting programming and execution of mobile agent based distributed applications, we provide a skeleton-based parallel programming environment, based on specialization of Algorithmic Skeleton Java interfaces and classes. Their implementation include mobile agent features for execution on heterogeneous systems, such as clusters of WSs and PCs, and support reliability and dynamic workload balancing. The user can thus develop a parallel, mobile agent based application by simply specialising a given set of classes and methods and using a set of added functionalities.

## 1. Introduction

The Mobile Agents model [22] has the potential to provide a flexible framework to face the challenges of Distributed Computing, especially when targeted toward heterogeneous distributed architectures. It represents an effective alternative to the Client-Server paradigm in several application fields such as e-commerce, brokering, distributed information retrieval, telecommunication services [1]. Several characteristics of potential benefit for scientific distributed computing can be provided by the adoption of the mobile agent technology, as shown in the literature [5,6,21]: they range from network load reduction, heterogeneity, dynamic adaptivity, fault-tolerance to portability paradigm-oriented development. Our research activity in this field concerned the application of the Mobile Agent paradigm to parallel and distributed systems programming, with particular focus to hybrid systems with hierarchical structure (such as clusters of SMP) [13,14,18] and Grid systems. It was studied the mobile agents paradigm integration with other paradigms, languages, primitives and libraries at low and high abstraction level, such as OpenMP [15], skeletons [19], Object oriented [18], message passing and threads, and optimized libraries, with application to real applications of highly irregular nature, such as N-Body and Branch & Bound [13,14]. A prototype tool, MAGDA [16], has been designed and currently developed. MAGDA is a framework for supporting the programming and execution of mobile agents [17]. It supplements

mobile agent technology with a set of features for supporting parallel programming on a dynamic heterogeneous distributed environment – as present in Grid systems. Here we are going to describe how the characterization of a mobile software agent overcomes its implementation and the task he has to carry out. We will introduce the model of its lifelines in our applications and how algorithmic skeleton will exploit this model in order to implement high level facilities to support parallel and distributed programming. Many authors presented taxonomies of distributed algorithmic skeletons [2,4,12]; in particular, we refer to Campbell's work that examined the classification of algorithmic skeletons and proposed a general one. According to this classification many kinds of algorithms can be included in the Farm-like and Divide and Conquer-like skeleton. Other works [12] use algorithmic skeleton approach mainly to separate the communication/synchronization structure from the application-specific computation in order to optimize the mapping of skeletons on different underlying distributed architectures. Similarly many programming environments [3,9,11], based on functional or OOP languages, have been developed to provide, through skeletons, implicit parallel programming mechanisms. A mobile agent based approach for Paradigm-Oriented Distributed Computing is described in paper [6].

## 2. Modeling agent in object oriented programming

A Software Agent is a program able to act autonomously, on behalf of its owner [7]. The characterization of an agent overcomes its implementation and the task he has to carry out. It can be defined as a tuple of properties such as name, origin, owner, state, server, context. A namespace should be defined in order to grant that agent instance is univocally identifiable among the ones who share the same code. The origin makes known the place were the agent was created, while the owner is the responsible for that agent. Through a delegation based mechanism the agent should inherit the owner profile in order to get the authorization he needs to perform its task. Agent state collects the results of execution. A server is the site able to host an agent. It accepts and handles creation, cloning, migration and disposal requests. It receives and forwards messages. It processes authorization requests granting access to local resources to incoming agents. A server is able to host an agent inside a context. The context defines the execution constraints for a group of hosted agents. This mechanism allows execution of agents on the same server but according to different policies and in separated rooms. When an agent travels across a network he needs to take with itself a set of information which makes possible its identification and its execution. The agent code is composed of a set of classes which implement agent's behavior. It can be transmitted together the agent state, or provided by an URL that specifies codebase, an archive file and a protocol to be used to fetch them. When we want to model an agent we have to characterize an object with its own behavior, we have to provide it with autonomy and reactivity as concurrent activities. This model allows us to define for a single agent many execution flows which are concurrently scheduled and are able to share the same data. They represents the agents behaviors whose execution depends by the agent status. Concurrency provides agents with a kind of parallelism by which it is possible to to react to different events and at the same time to pursue some goals. By a comparison with men's activities, we can design an Agent Model, whose proactivity and reactivity proceed together and influence each other.

### *2.1. Lifeline of an agent*

We intend to describe the lifeline of an agent and components which implement its behavior. Autonomy, reactivity and proactivity can be implemented by providing agent's model with a set of mechanisms which are common to the men's lifeline. A man is always looking for carry out an established plan of activities.

For example in a day he spends some time on its job, part of its time on its favorite hobbies, other with its family or friends. He is proactive to achieve his goals. Other activities in the man's life are executed as a reaction to internal or external stimuli. On the basis of internal stimuli, he eats when he is angry, or he sleeps when he is tired. By the external ones, he could decide to have a walk when it is sunny and to read a book when it is raining. These activities are scheduled according to man's priorities which depends on his desires and motivations. A men could feel these sensations when some time has elapsed without to eat or to sleep. These are external stimuli which condition his behavior. The model we propose is composed of the following kind of execution threads that characterize the behavior of an agent:

– Main proactive thread
– Listeners
– Timers
– Context listeners
– Communication thread

These activities, which are carried out to achieve a specific goal, correspond to autonomous and proactive behaviors, which a software agent should have. In order to provide software agents with reactivity we need to add some listeners, which are able to wait for events and to activate new behaviors, or to influence the ones which are already running. Listeners are able to detect some events, to interrupt proactive behaviors and to execute the reaction to the event. In our model these listeners can be other threads which compose the agent lifeline. Of course we tell about threads but the concurrency can be handled at different levels: by the agent itself, by the platform, by the multi-threads scheduler of the underlying system. In our model a priority mechanism allows to schedule the proactive tasks and the reactive routines according to a designed policy. The priority values could dynamically change according to the agent's status. Reaction of agent can be triggered by internal or external stimuli. About internal stimuli some kinds of events can be related to expiration of timers. Otherwise internal stimuli can be connected to internal believes of an agent. A change of agent's internal status can affect active behavior, but also can start new ones (for example the belief "had a coffee" can activate the behavior "have a cigarette"). On the other side external stimuli are generated by some events in the context where the agent resides. Some listeners should be added in order to manage the events which are generated in the context. The agent has to subscribe its listeners to the context and context will provide to wake up them on the occurrence of each event that the agent aims to handle. Social behavior is the last feature that characterize the agent's lifeline. Communication is a further component of the agent's behavior that can cause the activation of a new thread to be handled. A new message is forwarded to a handler which is able to start the specific routines. The lifeline of agent's model is illustrated in Fig. 1. Internal methods belong to the object, but do not define behaviors. Proactive threads represent those behaviors which are executed in order to get some agent's goals. Message handlers implements reactive behavior which perform the interaction with other agents when some messages are received. Listeners are able to detect events in the environment and to modify agent's beliefs or to start new actions.

### *2.2. The programming model*

The programming model of an Agent, according to the description we have introduced before can, be built on a set of APIs, which allow users to implement agent's behavior and to define its status. We can find different implementations which support the design and management of concurrent behaviors. Class abstraction or threaded routines are alternative solution. Scheduling of concurred behaviors could be managed by the programmer himself, or can be supported by the agent implementation or by the

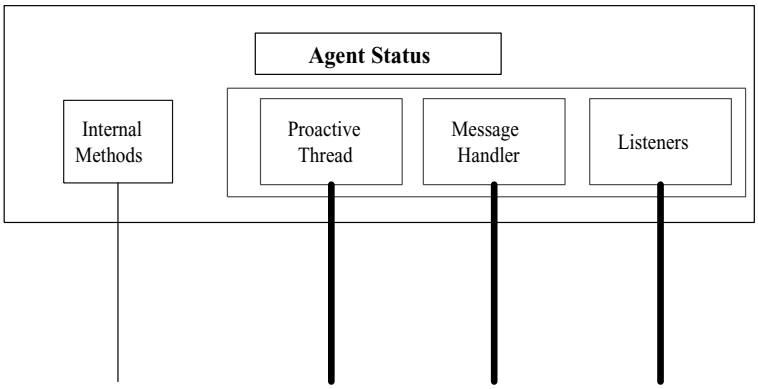


Fig. 1. Agent life cycle.

run-time environment. An agent class defines the parameters that characterize its status and overrides the methods which implement its proactive behavior.

In our environment the agents are provided with mobility. This capability makes necessary to address many issues. The first one deals with the persistence of the agent's status. In our model agents implements the weak mobility mechanism. It means that the dispatching of an agent requires the serialization of only the attributes of the agent's status and their transmission. At destination the application is resumed, not the executing process. Definition of agent's status, that is the choice of transient (the ones which cannot or must not be dispatched) and not transient data, among the class attributes, is the first not trivial task carried out by programmer. Some examples of not serializable data are sockets or files which are necessary linked to the physical machine executes. The address of the hosting machine is another parameter that is part of the agent status, but it changes its value each time the agent is dispatched to another destination.

*2.2.1. The main proactive thread*

As we said before this thread carries out the proactive behavior of an agent. A programmer is able to describe the main agent's task overriding a specified method inside the agent class. This method corresponds to the run routine of main agent's thread and it will be restarted each time the agent's status has been resumed. This model deals with a weak mechanism that does not allow the resuming of threads inside a process. The degree of autonomy of an agent allows him to activate or deactivate its listeners, to set events in the context, to send messages to other agents or to set its own internal events or the ones of other agents. An agent can choose to move by itself to another host, to create other agents or to die. The most important feature of an agent is autonomy that is granted by the programming model. It is essential that when an agent is created any other program can not own its control. It is possible by preventing that creation of an agent returns a memory reference to the agent instance.

*2.2.2. Events and listeners*

According to the programming model we are exploiting, an agent is able to catch two kinds of event. Events belonging to a first category are related to lifeline of agents (creation, cloning, migration, message reception). Second ones occur inside the execution context (pending shutdown, agent arrival, new agent creation, ...). To react to these events some listeners must be defined. A listener is an animated object itself. Its class is provided with the set of API that builds the Agent Programming Environment. A listener waits for an event and implements its handler. When a new event occurs, a listener is able to

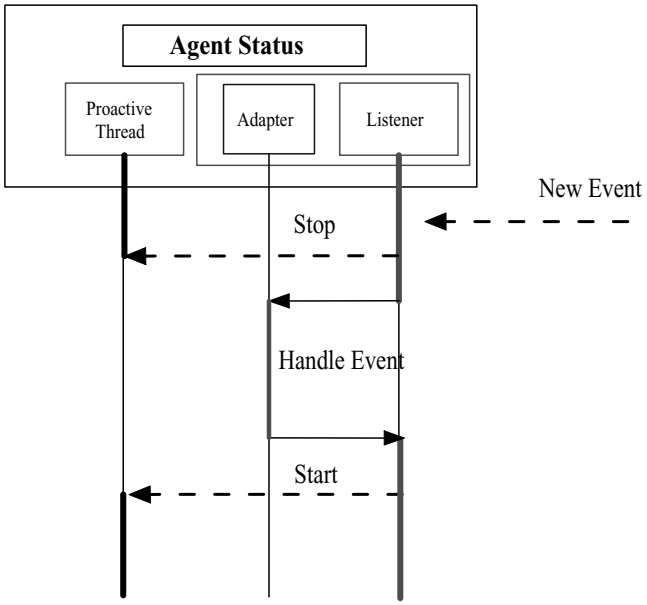


Fig. 2. Listener model in the agent.

stop the proactive thread and to access the agent's status. Two parameters must be provided in order to initialize a listener:

– The event to be handled
– The Adapter for handling

The Adapter is an interface, which is implemented by the programmer in order to describe the behavior of a listener, and describes the reaction to a particular event. In Fig. 2 is illustrated the listener model. An example can be a mobility event that is generated when an agent migrates from an host to another one. A mobility Adapter can be used to define the behavior of an agent before dispatching and on arrival. A special listener is the Message Handler. It handles incoming messages. Handling a message is quite different from handling an event. A message is characterized from additional parameters, which are:

– A sender
– A kind
– A content

An agent can know, by the kind value, if it is able to handle that message and how to handle it. The sender parameter allows an agent to personalize the response and to send it back. In our model when a message is received an answer is always due. The simplest response is a notification of reception and the result of message handling (successful or unsuccessful), otherwise the response could be a message itself.

## 3. Mobile Agents and the distributed object programming paradigm

The *Distributed Objects* (DO) paradigm represents a further development of conventional object oriented programming to support development of parallel and distributed applications capable of seamlessly

utilizing heterogeneous computing resources ranging from small-scale Cluster computing to large scale Grid computing. Whereas the mobile agents paradigm targets mostly loosely connected applications, the distributed objects paradigm focuses on a more tight interplay among application components (although they can be widely distributed). The main feature which makes an agent different from a distributed object is autonomy. An object, after a creation or a dispatch action should be activated by a method invocation to start its execution. As we said above the memory reference of an agent is not available to the programmer in order to grant agent's autonomy. In this way an agent can choose to reject external requests for services, such as cloning and mobility. In DO programming the interaction among objects is supported via remote method invocation mechanism by synchronous, asynchronous and one-side method invocation. A sending object can transfer data to a receiving object via any method invocations. Whereas a receiver could use synchronous (blocking) or asynchronous (non-blocking) method invocation to get the data from a source (without the need of an explicit sender).

MA paradigm provides instead a set of Java API designed for inter-agents communication. An agent can send a message object to another agent using asynchronous, synchronous or *future* send primitives.

Both approaches require an active server which run on target nodes, which are used to distribute the program. Servers provide an environment to agents and to objects for their execution. Furthermore they provide connections with other servers, which run on different nodes, system parameters, and some basic services such as deactivation/activation , dispatching and retracting, cloning and disposing. A set of events are notified by the server to running agents, providing them with the possibility to react to changes in the execution environment such as the shutdown of the hosting node. Above all in mobile agent programming is relevant to provide the possibility to handle security and other policies about agents permissions and administration.

## 4. Mobile Agents paradigm to support a skeleton-based programming approach

After all, using mobile agent paradigm remains a non trivial task because mobility introduces additional difficulties in designing coordination, synchronization and communications among different tasks. In most cases, especially when starting point is an available sequential code, utilization of high level interaction models, provided by some customizable skeletons, can ease the programming task, and the mapping of a parallel application on parallel systems. Algorithmic skeletons can be built as specialization of an agent's lifeline. A specialized model eases the task of a programmer providing a limited set of interaction models implemented in advance. We integrated algorithmic skeletons in a framework for supporting programming and execution of mobile agent based distributed applications, the *MAGDA* (Mobile AGents Distributed Applications) toolset [17]. We have implemented a set of Java packages, which enable to program distributed applications by adopting a skeletons-like approach [19], exploiting peculiar features of both Object Oriented and Mobile Agents programming models. By means of available skeleton interfaces a programmer is able to implement its own application by specializing an assigned structure and using a set of functionalities that a mobile agents framework can offer. A predefined algorithmic skeleton allows to reuse the sequential code by filling some methods, classes and interfaces and hides the difficulties involved by an explicit parallel programming paradigm. Difficulties and features of the Mobile Agent programming paradigm can be managed at a lower level, transparent to the user. Two algorithmic skeletons involving the *Farm*-like programming paradigm and the *Divide and Conquer*-like programming paradigm respectively have been implemented and tested. In *Processor Farm* programming paradigm a master process creates a number of slaves and assigns some work to everyone of them. The slaves compute their work and return the results to the master. *Task Queue* is

the most general *Farm*-like skeleton, every slave may produce new work to be performed by itself or by other slaves. The second algorithmic skeleton we have implemented belongs to *Divide and Conquer*-like skeleton class. It is an example of *Tree computation* algorithmic skeleton. It solves the initial problem dividing it in several subproblems assigned to different agent workers. Data flow from the root into the leaves and solutions flow back up toward the root. We chose to implement a binomial algorithm to build our tree, so its shape and results recombination procedure is consequentially determined. In the following we show how design particular agents behaviors in order to implement the *Processor Farm* and *Tree computation* skeletons introduced above.

## 5. The tree computation skeleton

### *5.1. Skeletons' description*

The skeleton frequently referred to as *tree computation* consists of a set of processes connected by communication channels according to a tree structure. Each process receives a problem to be solved, tests for a condition and then either splits the problem into k subproblems that are sent to k child processes, or does some processing work. When a process terminates its job, it remains waiting for a reply from each child, then combines these replies and sends back new result to its parent. In other words, according to this skeleton, data flow from the root into the leaves and solutions flow back up toward the root. In practice a number of questions (how to choose type, degree and depth of the tree, number of processes to be created, etc.) have to be answered before a working program can be produced out of this skeleton. However, they do not depend on the particular nature of the computation to be parallelized, but, instead, they are part of the skeleton and can be solved once and for all in the context of the skeleton itself. Our implementation mainly follows the general structure depicted in previous section, except that processes are replaced with agents and we have chosen to implement a binomial algorithm to build our tree, so its shape and results recombination procedure is consequentially determined. Here we have an agent that can be at the same time a son and a father. Proactive behavior is implemented by the *run*() method. Reactivity of agent to incoming messages is implemented by the *handleMessage*() method. The lifeline of the agent evolves as follows. When the agent starts, it is a son. At beginning its main goal is to become a father, so in the main thread it clones itself. Next goal to be scheduled is to solve the problem assigned to it. Then the agent needs to collect results from its sons. The agent asks each son for its result and waits for reply. The reaction to a new message from a son collects and reduces received results. When all results have been collected the second goal is achieved. Agent behaves again as a son and waits for a joining request from its father. When a new message has been received he reacts by returning its results and disposing.

### *5.2. Implementation in MAGDA*

Initially, the first agent of application starts, cloning itself and generating a first son, splitting and assigning half of initial workload to it. At each following step, every agent clones itself once, thus generating a son; at each step all agents simultaneously clone themselves, until the number of agents matches a target number. The tree shape and the communication pattern among agents are consequentially determined, as is shown in Fig. 3. The number of levels of the tree is equal to the number of cloning steps. The number of nodes belonging to the i$^{th}$ level is equal to $2^{nc-i}$, where *nc* is the number of cloning steps and *i* is the level index. A the end of this cloning step each agent can start solving its subproblem.

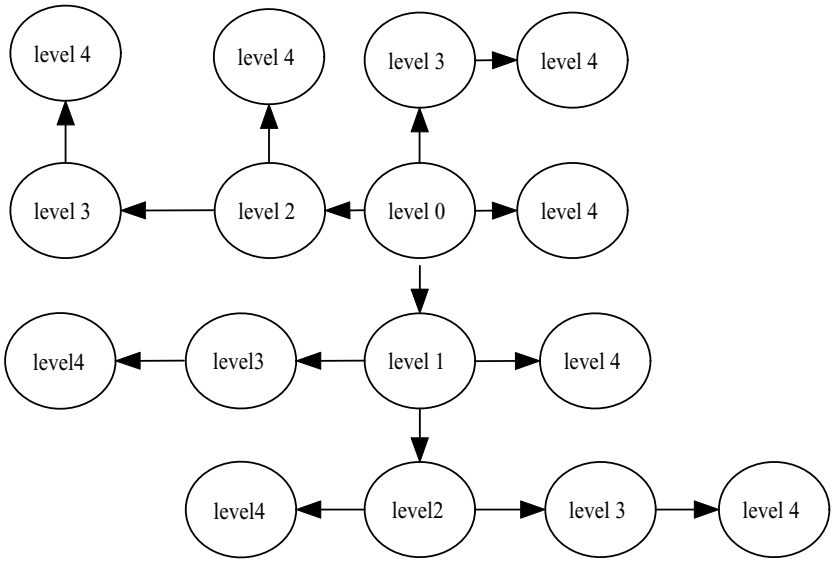


Fig. 3. Cloning/communication graph of the Tree Computation skeleton (binomial tree case).

Once the work is performed each agent collects and joins the results from its sons (from the youngest to the oldest one) and in its turn pass them to its parent. An agent returns its results only if it is not a parent or it has collected the results from all its own sons (from the youngest to the oldest one). We have defined a *BinomialTree* class which performs the basic actions of described skeleton behavior and activates specific methods that the user should override in order to specialize agent's behavior. All the names of such methods begin with the word $user\_$. The following methods belong to the *BinomialTree* class and should be overridden:

- The *user_init*() method, invoked by the first agent in order to initialize the application data.
- The *user_onCloning*() method, invoked in order to generate a new agent.
- The *user_Solve*() method, invoked in order to solve the agent's task.
- The *handler_message*() method, invoked in order to handle a message from the application.
- The *user_joinToFather*() method, invoked by an agent in order to return its results to its parent.
- The *user_fatherJoining*() method, invoked by an agent in order to join its own results to its sonŠs results.
- The *user_Close*() method, invoked, by the first agent, on occurrence of its disposing, at the end of the application, in order to present final results.

In Fig. 4 a pseudo-code is shown, sketching the skeleton's implementation within MAGDA framework. The first agent begins its execution by reading skeleton's input data, and user data (by calling the overridden method *user_init*()). Cloning phase requires the specialization, according to the specific application, of data and of behavior for both parent and its generated son. This is performed by means of *user_onCloning*() method to be overridden. After it has been generated, the son dispatches itself to its destination and at the end of the cloning phase it begins to compute its own data: the user's overridden method called by the framework is the *user_Solve*() one. At the end of the computational step every agent has to join itself to its sons in reverse order with respect to the generation; in order to perform the joining phase the framework calls the *user_joinToFather*() methods for the son agent and the *user_fatherJoin*() method for the parent agent. Only the first agent of the tree will call the *user_Close*() method to present final application results.

```
OnCreation(){user_init();}

run(){
 if(!dispatched)    dispatch(myDestination);
 while(clones<nodes){
   clones=2*clones;
   user_onCloning();
    clone();
 }

 user_compute(myObject);

 while(!mySons.isEmpty()){
   next=mySons.next();
    next.send("jointofather");
 }
 wait("jointofather");

  if(!master)  dispose();
  else user_Close();
   dispose();
}


handleMessage(Message m){
   if(m.sameKind("jointofather"'))    myFather.reply(user_joinToFather());
    else user_fatherJoin(m);
}
```

Fig. 4. BinomialTree class pseudo-code.

### 5.3. An example: Quick sort

We show now, through a simple example, how the above described class and methods can be used in order to develop a distributed mobile tree computation. Chosen computation is the simple and well known Quick Sort algorithm, applied to an array of integers. In the user-defined *QuickSort* class, user declares and initializes all data that each agent needs in order to solve its own sub-problem: in the QuickSort example data are an array of integers to be sorted, and the sub-arrays' bounds, by means of which each agent is able to know which part of the original array is assigned to. The first agent fills up its array by calling the user-overridden $user\_init()$ method and sets its lower bound to 0, and the upper one equal to the length of the array. The $user\_onCloning()$ method calls the user-defined $split()$ method, in order to split the array into two sub-arrays, representing parent's sub-problem and the one of its son. The method returns two new QuickSort objects for the two agents (the parent and the son). When the cloning phase ends up, user-overridden $user\_Solve()$ method is called, performing a recursive sorting of the array (by calling the $sort()$ method of the $SeqQuickSort$ object).

In this example the $handler\_message()$ method is not used because the sorting computation doesn't require communication among agents. The $user\_joinToFather()$ method is called by son when its parent is collecting the expected results: in QuickSort example this method returns a sorted sub-array. The $user\_fatherJoining()$ method is called by parent its son's results have been received; in the example this method performs the fusion between the array owned by the current agent and the ones returned by the sons. The $user\_Close()$ method finally prints the sorted array.

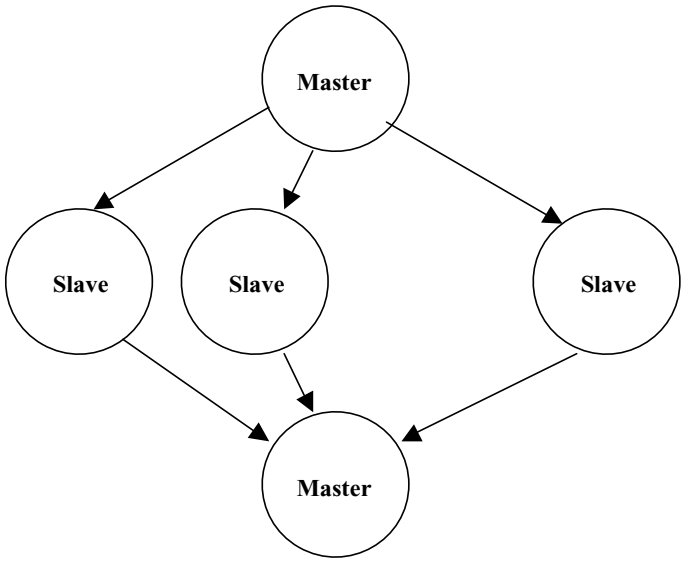


Fig. 5. The communication graph of the Task Queue Skeleton.

## 6. The task queue (Processor Farm) skeleton

### *6.1. Skeletons' description*

Skeleton usually referred to as *processor farm* consists of a *coordinator* process and a set of *worker* processes that act as slaves. In a processor farm, the coordinator decomposes the work to be done in subproblems and assigns a different subproblem to each worker. Upon receipt of a subproblem, each worker solves it and returns a result to the coordinator. Again some details have to be defined before skeleton can become a working program, and slightly different organizations can be selected for the processor farm (for instance workers may or may not be allowed to communicate each others). However, even in this case, these issues concern the skeleton definition and can be entirely dealt with in the skeleton context. We have chosen to implement the *Task Queue* Skeleton, that is the most general *Farm*-like skeleton: every slave may produce new work to be performed by itself or by other slaves. The task queue skeleton is illustrated in Fig. 5.

### *6.2. Implementation in MAGDA*

Here we have to define two different agent behaviors and their interaction protocol. Master agent goals are: an initial distribution of tasks, reduction of results and notification of termination to the slaves. It is proactive in reading the input data, dividing the problem in a certain number of subtasks and to verify the termination condition and notify it. These actions will be executed in the agent main thread. The collection of results provided by the slaves is part of the reactive behavior that is activate when a new message has been received. A slave is an agent whose goal is to compute the task and return the result to its master. All these action are carried out in the main thread. Slaves also look for new tasks to be solved, which can or cannot be available in any moments. A listener that is waiting for a message implements agent's reaction to the arrival of a new task to be solved, or to a disposal request, which has been sent by the master when all tasks have been solved. In the following section our single program

implementation of the agent's behavior is described. For this skeleton the MAGDA framework makes available the *TaskQueue* Java class, and the *Task_Interface* interface. TaskQueue class describes the behavior of an agent worker and of its master. When an user extends this class it needs to declare and initialize all data that each worker needs in order to solve the generic task. The user has to implement its worker by extending the class and overriding the following methods:

- The *user_init*() method, invoked by the first agent worker in order to initialize the application data;
- the *user_split_task*() method, invoked by the first worker in order to fill the bag with more subproblems, which will be distributed among all the workers;
- the *user_handle_message*() method, invoked in order to handle a message from the application;
- the *user_Close*() method, invoked by the first agent worker at disposal, in order to present final results.

In Fig. 6.2 a pseudo-code is shown, sketching skeleton's implementation within the MAGDA framework. The first agent begins its execution by reading skeleton's input data, and user data (by calling the overridden method $user\_init()$). The splitting of the initial problem is performed by the user, by overriding the $user\_split\_task()$ method. Effect of this phase is filling agents' queue with a certain number of subproblems. Then the master clones itself, distributes the queue among different clones and dispatches the workers to the target hosts. All global data are cloned together with agents. Arrived at destination each agent activates a thread in order to extract and compute those tasks, which are stored into the queue. When the queue is empty a request for new tasks is sent. If no more tasks are available the worker signals the *empty queue condition* to the master. The master, after its computation ends, receives the empty queue signal from all workers and check the termination condition by calling the user-overridden $user\_stopCondition()$ method. If a true value is returned all workers can be disposed and the master can terminate itself. When a message is received, a message handler start asynchronously. The handler reads the message tag and if it is not able to handle it, because it is an application specific message, calls the user-overridden $user\_handle\_message()$ method. In order to specify how a task extracted from the queue has to be consumed, the user must define a $Workload$ class that implements the $Task\_Interface$ interface. Task_Interface is implemented as follows:

```
public interface task_interface{
    public void compute();
 }
```

The $compute()$ method implemented by the user is called by an agent worker on the user's object in order to complete the task. Other methods, which need not to be overridden and can be used by the programmer, are defined in the $TaskQueue$ class. As the skeleton is designed in order to hide the real distribution of the application, the programmer is not able to locate each single worker. It means that only collective communication are allowed, in order to share global values and update them, or in order to synchronize all the workers. The $procFarmBroad(Message)$ method allows the programmer to send a message to all the other workers. The *bag_push*(*Object obj*) method allows the programmer to insert some new tasks into the queue according to the Task Queue skeleton.

### *6.3. An example: A combinatorial optimization application*

We show, through a quite complex real application, how the above described skeleton's classes and methods can be utilized. The chosen application is a combinatorial discrete optimization, performed with

```
OnCreation() {
 master=true;
 init_data_par(); //input parallel data
 user_init(); //input application data
 user_task_split (int n); //problem splitting
 agents_split (); //agent splitting and dispatching
 }

run() {
  while(!stop) {
   do{
      runThread.start();
      waitThreadEnd();
      repeat= balance();
     }while(repeat);

   if(!master){
     sendMessage("empty_bag");
    waitMessage();
   }
   else{
    while((!stop)&(bag.empty()){
      waitMessage();
      stop=user_stopCondition();//all queues are empty
     }
    }
   }//end while
   user_Close();
    sendToAll("Dispose");
    dispose();
}//end run

handleMessage(Message msg){
  if(msg.sameKind("service msg") {...}
   else  user_handleMessage(msg);
}

Class runThread {
run(){
  while(!bag.empty()){
   newTask=(Task_interface) bag.pop();
   newTask.solve();
  }
 }}
```

Fig. 6. TaskQueue class pseudo-code.

the well known Branch and Bound method. A *discrete optimization problem* consists in searching the optimal value (maximum or minimum) of a function $f : \vec{x} \in \mathcal{Z}^n \rightarrow \mathcal{R}$, and the solution $\vec{x} = \{x_1, \ldots, x_n\}$ in which the function's value is optimal. $f(\vec{x})$ is said *cost function*, and its domain is generally defined by means of a set of $m$ constraints on the points of the definition space. Constraints are generally expressed by a set of inequalities:

$$\sum_{i=1}^{n} a_{i,j} x_i \leqslant b_j \qquad \forall j \in \{1, \ldots, m\} \tag{1}$$

and they define the set of feasible values for the $x_i$ variables (the *solutions space* of the problem). Branch & Bound [8] is a class of methods solving such problems according to a *divide & conquer* strategy. The initial solution space is recursively divided in subspaces, until attaining to the individual solutions; such a recursive division can be represented by a (abstract) tree: the nodes of this tree represent the solution subspaces obtained by dividing the parent subspace, the leaf nodes represent the solutions of the problem, and the tree traversal represents the recursive operation of dividing and conquering the problem. The method is enumerative, but it aims to a non-exhaustive scanning of the solutions space. This goal is achieved by estimating the best feasible solution for each subproblem, without expanding the tree node, or trying to prove that there are no feasible solutions for a subproblem, whose value is better than the *current* best value. (It is assumed that a best feasible solution *estimation function* has been devised, to be computed for each subproblem.) This latter situation corresponds to the so called *pruning* of a search subtree. The Task queue skeleton is thus very well suited to a parallel implementation of the Branch and Bound method. According the skeleton's description given above we have developed two java classes: the *BB_Application*, which extends the *TaskQueue* main class, and the *BB_Task* class, which implements the *Task_Interface* interface. In the example the class data are the two arrays of weights and the matrix of bounds. The first array is used in order to compute the best value; the second one and the matrix are used to check if the bounds are satisfied. The local best value and the corresponding solution are member variables too. The *user_split_tasks*() reads from a file the elements of the arrays of data described before. In the B&B implementation the *user_split_tasks*() method fills the queue with a number of tasks greater or equal to the number of workers. Each task is a node belonging to the same level of the tree of solutions. In B&B implementation *handler_message*() method updates the optimum value that is communicated by the workers through a collective communication. The *user_Close*() method prints the final optimum value and the correspondent solution. The *BB_Task* class implements the Task_interface, which is used by the worker in order to manage the single task of the bag. An object of this class declares only those data that are local to the single task. In B&B implementation the data are:

- *int sc[]*: an array of integers, which represents a subset of solutions, (a node of the tree),
- *int lsc*: an integer, which represents the level of the tree where the above node is placed.

The *compute*() method, is invoked by the worker in order to solve the task. The *compute*() method reproduces the sequential code for the visit of the tree. It starts from the node correspondent to $sc[]$ array, of the $BB\_Task$ object extracted form the bag. When a node of the tree must be processed later a new $BB\_Task$ object is built, and it is pushed in the worker's queue. When the leaf of the node is reached the task is computed and another one will be extracted from the queue by the worker.

## 7. Experimental results

In this section we discuss experimental results of execution of the mobile agent based branch and bound optimization application described in the previous section. The target architecture is a cluster of 8 Pentium Celeron processors, 800 MHz clock frequency and 128 M RAM, connected via an 100 Mb/s ethernet switch. Experimental figures are provided, with varying the problem size and the number of computing nodes. Because of the highly irregular behavior of this kind of algorithms, due to the branching effect, the simple speedup measure is meaningless for our purposes. The distributed branch biasing effect can be taken into account by considering the total number of solved subproblems, which is largely variable with different executions, even with the same input data. We have thus defined $PRi$, as being the ratio between the number of solved problems during the parallel execution of the

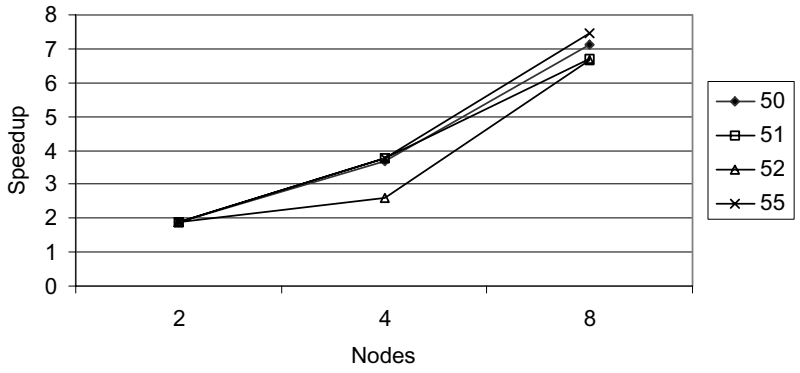


Fig. 7. Normalized speed up values for four different problem sizes.

application and the number of problems solved by the sequential version of the application. We then define a *normalized speedup* as being $Sn = Speedup * PRi$; this latter is considered in order to filter the branching effect in experimental results. In Fig. 7 the normalized speed up values are shown, for four different problem sizes, and from 2 to 8 processors.

The normalized speedup scales well with the number of processors for all the considered problem sizes.

## 8. Conclusions

We described the mobile agent programming model that was adopted in our research activities in addressing distributed systems programming. In order to overcome some additional challenges, which have been involved by the adoption of the proposed approach, we developed a framework that provides the programmer with high level programming skeletons. Skeletons implement automated mechanisms for agents' interaction exploiting migration and cloning. Algorithmic skeletons are presented as a specialization of the agent lifeline using some facilities provided by the exploited programming model. They allow to reuse parts of the sequential code by filling some methods, classes and interfaces, and to hide the difficulties to be faced with using an explicit parallel programming paradigm. A synergic integration, in the proposed approach, of the OOP concepts, the portability characteristics of the Java language and the features of mobile agent model. These peculiarities ensure a good programming easiness, by allowing the reuse of large portions of sequential code, and at the same time, don't prejudice the performance aspects, thanks to the highly dynamic adaptability of the implemented skeletons to the underlying architecture.

## References

[1] D.B. Lange, M. Oshima, *Programming and Deploying Java Mobile Agents with Aglets*, Addison-Wesley, 1998.
[2] D. Campbell, *Towards the Classification of Algorithmic Skeletons*, Tech. Rep. YCS-276, Dept. of Comp. Science, Univ. of York, 1996.
[3] F.A. Rabhi, Exploiting Parallelism in Functional Languages: a Paradigm-Oriented Approach, in: *Abstract Machine Models for Highly Parallel Computers*, Oxford University Press, April 1993, pp. 118–139.
[4] G.R. Andrews, Paradigms for Process Interaction in Distributed Programs, in: *ACM Computing Surveys* **23**(1) (March 1991), 49–90.

[5] R. Gray, D. Kotz, S. Nog, D. Rus and G. Cybenko, Mobile agents: the next generation in distributed computing, in: *Proc. of Int. Symposium on Parallel Algorithms/Architecture Synthesis*, IEEE Computer Society Press, 1997, pp. 8–24.
[6] H. Kuang, L.F. Bic and M. Dillencourt, Paradigm-oriented distributed computing using mobile agents, in: *Proc. of. 20th Int. Conf. on Distributed Computing Systems*, IEEE Computer Society Press, 2000, pp. 11–14.
[7] H.S. Nwana, Software agents: an overview, in: *The Knowledge Engineering Review*, (Vol. 11)(3), Cambridge University Press. 1996, pp. 205–244.
[8] H.W.J. Trienekens, *Parallel Branch & Bound Algorithms*, Ph.D. Thesis at Erasmus Universiteit-Rotterdam, Nov. 1990.
[9] J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu and R.L. Whie, Parallel Programming Using Skeleton Functions, in: *Parallel Architectures And Languages, PARLE'93*, LNCS 694, Springer-Verlag, 1993, pp. 146–160.
[10] M. Bull, M. Westhead, M. Kambites and J. Obdrzalek, Towards OpenMP for Java, in: *Proc. of* $2^{nd}$ *European Workshop on OpenMP – EWOMP'2000*, Edinburgh, 2000.
[11] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti and M. Vanneschi, *The* $P^3L$ *Language: an Introduction*, Technical Report HPL-PSC-91-29, Hewlett-Packard Laboratories, Pisa Science Centre, Dec. 1991.
[12] Murray Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press & Pitman, 1989.
[13] R. Aversa, B. Di Martino, N. Mazzocca and S. Venticinque, Mobile Agents for Distributed and Dynamically Balanced Optimization Applications, in: *High Performance Computing and Networking, Lecture Notes in Computer Science*, (Vol. 2110), Springer-Verlag, 2001, pp. 161–170.
[14] R. Aversa, B. Di Martino and N. Mazzocca, Restructuring Irregular Computations for Distributed Systems using Mobile Agent, in: *Applied Parallel Computing, Large Scale Scientific and Industrial Problems, Lecture Notes in Computer Science*, (Vol. 1947), Springer-Verlag, 2001, pp. 223–232.
[15] R. Aversa, B. Di Martino, N. Mazzocca, M. Rak and S. Venticinque, Integration of Mobile Agents and OpenMP for programming heterogeneous clusters of Shared Memory Processors: a case study, in: *Proc. of* $3^{rd}$ *European Workshop on OpenMP – EWOMP'2001*, Barcelona (S), September 2001 .
[16] R. Aversa, B. Di Martino, N. Mazzocca, S. Venticinque, MAGDA: a software environment for Mobile AGent based Distributed Applications, in: *Proc. of* $11^{th}$ *Int. Conference on Parallel and Distributed Processing*, IEEE CS Press, 2003, pp. 332–338.
[17] R. Aversa, B. Di Martino, N. Mazzocca and S. Venticinque, MAGDA: A Mobile Agent based Grid Architecture, in: *Journal of Grid Computing*, Springer Netherlands, (Vol. 4)(4), December 2006, pp. 395–412.
[18] R. Aversa, B. Di Martino, T. Fahringer and S. Venticinque, On the evaluation of the Distributed Objects and Mobile Agents programming models for a distributed optimization application, in: *Applied Parallel Computing. Advanced Scientific Computing, Lecture Notes in Computer Science*, (Vol. 2367), Springer-Verlag, 2002, pp. 233–242.
[19] R. Aversa, B. Di Martino, N. Mazzocca and S. Venticinque, in: *VECPAR'2002. 5th International Conference on High Performance Computing in Computational Sciences 2002. Selected Papers and Invited Talks.*, LNCS 2565, Springer, Berlin (2003), pp. 622–634.
[20] R. Aversa, B. Di Martino, N. Mazzocca and S. Venticinque, MAGDA: a software environment for Mobile Agents based Distributed Applications, in: *Parallel, Distributed and Network-Based Processing*, IEEE Computer Society Press, 2003, pp. 332–338.
[21] T. Drashansky, E. Houstis, N. Ramakrishnan and J. Rice, Networked Agents for Scientific Computing, in: *Communications of the ACM*, (vol. 42)(3), March 1999, pp. 48–54.
[22] V.A. Pham and A. Karmouch, Mobile software agents: an overview, in: *IEEE Communications Magazine*, (Vol. 36)(7), IEEE Computer Society Press, 1998, pp. 26–37.

---

**Salvatore Venticinque** received the M.S. degree (magna cum laude) in Informatics Engineering in 2000 and his PhD in "Electronic Engineering" from the "Seconda Università di Napoli" in 2003.

He is Assistant Professor at Department of Information Engineering of the Second University of Naples. He teaches "Computer Programming" and "Computer Architecture" in regular academic courses. He is involved in research activities dealing with Parallel and Grid Computing and Mobile Agents Programming for distributed systems.

He is author of publications in international journals, books, and conferences, in collaboration with national research organizations and foreign academic institutions (ENEA, University of Vienna, . . .).

He participated to research projects supported by national and international organizations. He has been co-chair of international conferences and member of several Program Committees.

**Rocco Aversa** graduated in Electronic Engineering at University of Naples in 1989 and received his Ph.D. in Computer Science in 1994.

He is Associate Professor (Assistant Professor from 1995 to 2004) in Computer Science at the Department of Information Engineering of the Second University of Naples.

His research interests are in the area of parallel and distributed systems. The research themes include: the use of the mobile agents paradigm in the distributed computing; the design of simulation tools for performance analysis of parallel applications running on heterogeneous computing architectures; the project and the development of innovative middleware software to enhance the Grid computing platforms. Such scientific activity is documented on scientific journals, international and national conference proceedings.

Rocco Aversa participated to various research projects supported by national organizations (MURST, CNR, ASI) and in collaboration with foreign academic institutions. In particular, in 2004–2006 he coordinated the Second University of Naples research group working in the project "Centre on Information and Communication Technology", supported by Regione Campania with a financing of more than one milion of euros. In 2005 he was appointed in the board of the directors of the consortium "Centro Regionale Information e Communication Technology" as the representative of the Second University of Naples.

**Beniamino Di Martino** received the M.S. degree (magna cum laude) in Physics and the Ph.D. degree in Information Engineering, both from University of Naples (Italy), in 1992 and 1996 respectively. Since 2005 he is Full Professor at the Second University of Naples (Italy). In 1994 he joined the Institute for Software Technology and Parallel Systems at the University of Vienna (Austria) where he was Researcher till 1998. In 1998 he moved at the Second University of Naples (Italy) where he was Assistant Professor till 2002, and Associate Professor till 2005.

He is author of 5 international books and more than 100 publications in international journals and conferences. He participated to various research projects supported by national and international organizations (international projects include: EU-IST OntoWeb and APART, EU-Esprit HPF+ and PPPE, CEI PACT, EU-TMR, Austrian-SFB AURORA, Austrian FWF HLPS).

He served as general and program chairman, and member in Program Committees, of several international conferences, and as guest editor for journals' special issues.

He is editorial board member and chair of international journals. He is member of the Executive Board of the IEEE CS Technical Committee on Scalable Computing.

His research interests include Mobile and Intelligent Agents, Semantic Web and Grid, Semantic based Information Retrieval and Text Mining, Natural Language Processing, Programming and Compiler Techniques for High Performance and Grid Computing, Parallel and Grid Architectures, Automated Program Analysis, Comprehension and Transformation.

**Nicola Mazzocca** is a full professor of "Sistemi di Elaborazione" (ING-ING/05 area) at the Computer and System Engineering Department, University of Naples "Federico II" From November 1998 to October 1991, he attended the Ph.D. program in Electronic Engineering and Computer Science (IV cycle) at the Computer Science Department of the University Federico II, Naples, major: Computer Science. In September 1992 he received the "Dottore di Ricerca" (PhD) degree in Electronic Engineering and Computer Science. Thesis: "Sviluppo ed Analisi di Applicazioni Parallele in Ambiente CSP". From 1994 to present, he taught over 30 university courses on topic ranging from computer organization, to high-performance systems, reliable systems, operating systems, computer programming. Prof. Mazzocca conducts his research activity at the Computer Science Department of the University Federico II, Naples, and at the ITC Regional Center of Competence, Regione Campania. His research activities are mainly centered on: computer architecture, dedicated systems, reliable systems, secure systems, distributed systems, high-performance systems, performance evaluation in high-performance systems. He authored over 170 papers on international journals, books, international conferences in the field of computing and computer networks. In the context of such activities he cooperated with numerous Italian and foreign institutions and universities. Such research activities led to proof-of-concepts and prototypes presented at international conferences and adopted by Italian and international research institutions and companies. From 1998 to 2004 prof. Mazzocca coordinated the Computer Science research group at the Second University of Naples. He took part in several research projects. During such projects, he was involved in research activities conducted by Italian and international institutions (Università di Torino, Firenze, Parma, CNR, University of Urbana, Caltech, University of Sheffield, JPL).