A Policy-based Mobile Agent Infrastructure

Rebecca Montanari[1], Gianluca Tonti[1], Cesare Stefanelli[2]

[1]*Dip. di Elettronica, Informatica e Sistemistica*
Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{rmontanari, gtonti}@deis.unibo.it

[2]*Dipartimento di Ingegneria*
Università di Ferrara
Via Saragat, 1 - 44100 Ferrara - Italy
cstefanelli@ing.unife.it

Abstract

The Mobile Agent paradigm seems to provide promising solutions for developing applications in the Internet environment. However, the adoption of mobile agents introduces specific problems related to the specification and control of agent migration strategies. The paper advocates a policy-based solution to support the flexible management and dynamic configurability of agent mobility behaviour. Our solution permits to define agent migration strategies at a high level of abstraction, separately from the agent code, thus promoting a clear separation between mobility and computational concerns. The paper describes the policy-based middleware that allows to adapt at run time agent migration strategies to evolving application requirements and environment conditions without impact on the agent code implementation.

1. Introduction

The widespread diffusion of the Internet and the recent advances in telecommunication and wireless systems are enabling a pervasive and ubiquitous computing infrastructure for service provision. In this scenario, users require services that can be flexibly customised to accommodate their needs and dynamically reconfigured to adapt to the characteristics of the current points of attachment and to the current conditions of the available computing infrastructure. In addition, they expect to access services independently of their physical location, e.g., at their workplaces, at homes, at public Internet kiosks.

To address these requirements new programming paradigms based on Mobile Code (MC) technology are becoming increasingly popular among researchers and practitioners for service design, development and maintenance [1]. A significant example that is enjoying a lot of popularity is represented by the Mobile Agent (MA) paradigm that proposes the mobility of a whole software component (both code and execution state) [2].

However, the adoption of the MA paradigm raises new challenges in the design and deployment of applications. The location where an MA executes becomes a first-class element to take into account in the application design. In particular, MA application designers have to explicitly express and decide when and where to move MAs on the basis of the knowledge of application and environment state.

The traditional programming approach to agent mobility is to statically hard-code migration rules into the agent code at design time. This lacks flexibility and cannot be appropriate in the new Internet scenario where a-priori assumptions on network topology and on the status and availability of resources and nodes are unlikely to hold at run-time. New solutions are required to increase the flexibility of mobility programming so that agent migration strategies can be easily modified to accommodate the run-time changes in the agent operating environment and in application requirements.

In this paper we present a policy-based approach to leverage the design and development of MA applications. Policies are rules governing choices in the behaviour of a system separated from the components in charge of their interpretation [3]. Policies simplify the problem of expressing complex management strategies and increase the flexibility and dynamicity of systems management [3], [4].

Our choice of adopting a policy approach to mobility

This investigation is supported by the University of Bologna (Funds for Selected Research Topics: "An integrated Infrastructure to Support Secure Services"), by CNR (Funds for Young Researchers: "An Integrated Security Infrastructure for Multimedia Services") and by MIUR within the framework of the Project "MUSIQUE: Multimedia Ubiquitous Service Infrastructure in a QoS Universal Environment".

follows the assumption that the underlying principles for policy-based systems seamlessly apply for mobility management: we use policies for specifying when, where and which agent parts have to migrate, externally to the agent code. The details on the policy-controlled mobility model are provided elsewhere [5].

This paper focuses on the middleware we have developed to simplify the design and deployment of policy-based MA applications in real application scenarios. In particular, the middleware provides application designers with support tools for facilitating the specification, update and reuse of high-level agent migration policies, separately from the application code. The same middleware simplifies runtime policy deployment: it automatically and transparently installs policies into MAs, activates desired changes in the agent migration patterns accordingly to high-level policy specifications and propagates possibly policy variations to interested MAs with no impact on their code implementation.

2. A Policy-based Model to Agent Mobility

Mobile agents are executable entities that can autonomously roam the network, along with their *code* part (that describes the computation to perform) and their *state* part (that contains the agent private data, the current execution state, and the set of references to external resources) [1]. Agent mobility can be either *reactive* or *proactive.* In the former case, agent migration is triggered by an external entity that has the management authority to decide agent mobility. In the latter case, it is the agent itself that autonomously determines when to migrate. In addition, there are several types of agent migration patterns at different levels of granularity. For instance, one MA could decide to migrate either with whole or only with parts of its code. The choice typically depends on dynamic deployment conditions. For instance, in a wireless scenario with frequent disconnection it could be convenient to send in one shot the whole agent code, while on a LAN it could be more convenient to move the core code part and to download the others on demand [6].

From these considerations it emerges that the space of choices that programmers have to face involves the definition of several elements, from the entity that is in charge of triggering the mobility strategy, to the entity that is targeted by relocation, from the conditions that trigger agent migration to the type of migration pattern to activate. The specification task of the agent mobility behaviour is further complicated by the dynamicity of the Internet environment that rules out the possibility to make predefined assumptions on the contexts of agent execution.

The traditional programming approach to MA applications provides poor programming abstractions and structures for mobility [7]. Computational and mobility aspects are typically mixed within the agent application code making difficult to reconfigure or introduce new agent migration patterns at run time without impacting on the application implementation. Recent research activities are recognizing the relevance of separating the application functional aspect from the mobility ones to reduce the complexity and to leverage the development of MA applications in different, statically unknown, usage scenarios [7], [8], [9].

Toward this goal, we advocate policies as a powerful concept to achieve separation of concerns. A policy-based approach to mobility requires to develop a comprehensive programming model to permit programmers to easily specify the mobility strategy of MAs, that could be possibly modified at run-time to adapt to changing execution conditions. At the same time, the model should permit to proceed from the mobility specification to a design sufficiently detailed to be implemented directly. Toward this purpose, it is essential to precisely define how to express and represent mobility policies, to decompose the application in well-defined building blocks to achieve complete separation between mobility concerns and application functionality and to structure them in a consistent and coherent manner. In addition, it is necessary to build a support infrastructure that can provide appropriate policy-enforcement mechanisms that manage changes in the agent mobility behaviour transparently to programmers.

2.1. Programming Agent Mobility

To facilitate the definition of mobility strategies we adopt the Ponder policy language, developed at the Imperial College [4]. In particular, we use a subset of the Ponder language, i.e. the Ponder obligation policies. Herein we report two simple examples of the use of Ponder for the management of agent mobility. Please refer to [5] for details on the specification on Ponder-based agent migration strategies. In Table 1 the policy named *P1* allows us to specify proactive agent migration. It states that the agent called *Manager* has to migrate to the G1 node when the current execution node becomes overloaded (the event *CPULoad(90)* following the `on` keyword). The migration action (the *go()* method) is triggered by a CPU usage exceeding 90% and is executed by the agent itself. The migration method takes the destination execution node (*G1*) and the name of the method to perform (*run()*) as input parameters. The migration action can be performed only if *G1* is reachable and has the resources needed for agent computation (as monitored by an underlying monitoring system)(the `when` clause).

The policy named *P2* of Table 1 differs from *P1* simply because it models a case of reactive agent mobility. The entity responsible for the migration decision is not the *Manager* agent, but the external entity called *Relocator*. The *relocate()* method is used by the *Relocator* to transfer the execution of the *Manager* agent to the *G1* node.

```
inst oblig P1
 on CPULoad (90)
 subject s = agents/Manager
 do s.go(G1.toString(), "run")
 when
  MonitoringSystem.getReachabilityStatus(G1)==true  and
  MonitoringSystem.hasResources(G1, resources)==true
```

```
inst oblig P2
 on CPULoad (90)
 subject s = system/Relocator
 target t = agents/Manager
 do s.relocate(t, G1.toString(), "run")
 when  MonitoringSystem.getReachabilityStatus(G1)==true
```

Table 1. Ponder Obligation Policies.

3. A Policy-based Mobile Agent Middleware

Figure 1 depicts the organisation of the integrated middleware for policy lifecycle management in mobile agent systems. The infrastructure is designed according to a layered architecture, with a large number of services at different levels. In particular, the upper layer services support policy specification, activation and enforcement. Lower layer services monitor application and environment state and notify mobile agents about changes relevant for their migration decisions.

More in detail, the upper layer includes the following services:

- the *Specification Service* permits programmers to edit, update and remove mobility policy specifications. The service also transforms high-level policy specifications into policy enforcement modules, i.e., suitable code that can be interpreted at run-time by the underlying agent infrastructure;
- the *Repository Service* stores all currently specified and enabled policies and can be queried to retrieve specific policies;
- the *Distribution Service* distributes mobility policies to relevant entities at both policy instantiation time and at any successive change. In particular, it distributes policies to the policy subjects, either mobile agents (in the case of proactive mobility) or agent system components (in the case of reactive agent migration);
- the *Policy Enforcement Service* is responsible for the enforcement of triggered obligation policies. In particular, at event occurrence the service is delegated to coordinate the retrieval and interpretation of triggered policies and to activate migration and mobility-specific management actions accordingly to policy specification.

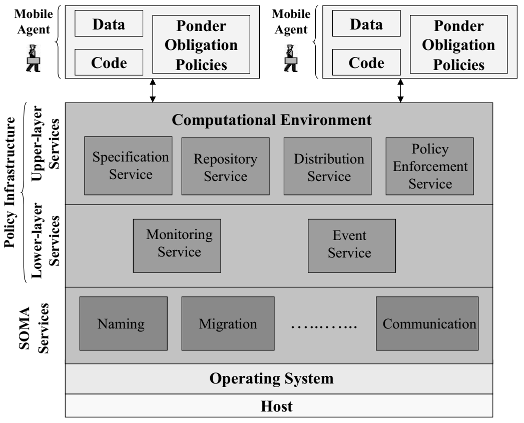

Figure 1. The Policy-based Infrastructure.

At the lower layer, the architecture provides the essential services to support the dynamic adaptation of agent migration strategies to changes in the execution context. The *Monitoring Service* detects the modifications in both application and environment state. The *Event Service* manages the events, i.e., the monitored variations in the whole distributed system, by interfacing with both agents and the Monitoring Service. More in detail, any policy subject can register its interest to one or more specific events with the Event Service. On its turn, the Event Service is responsible for dispatching registered events to interested entities. The Event Service typically receives state information from the Monitoring Service. As a key feature, the Event Service is designed to take into account agent mobility in order to notify events to interested agents even in case of migration.

The infrastructure services are integrated on top of the SOMA mobile agent system to exploit SOMA support services for agent naming, communication, migration, security and interoperability [10]. SOMA offers different locality abstractions to describe possible patterns of system interconnection, from simple intranet LANs to the Internet. Any SOMA fixed node is modelled by a *place* that provides the execution environment for mobile agents. Several places are grouped into *domain* abstractions that usually correspond to network localities.

3.1. Policy Specification and Initialisation

The *Specification Service* integrates the policy definition and management tools offered by the Ponder programming environment [11]. In Ponder a user interface assists in text editing of Ponder policies; a policy compiler transforms high-level Ponder policy specification into Java objects that act as simple policy information containers and that can be used across heterogeneous agent system platforms; a viewer helps in the efficient manipulation and effective navigation across the set of specified policies; a syntactic and semantic verifier controls new/updated Ponder policy specifications.

It is worth stating that in the current implementation of the policy-controlled model differentiated types of users can specify migration policies for MA applications. Application developers could specify at design time an initial set of default migration policies for MAs that can be enlarged or modified successively at deployment time by the users of MA applications depending on their application requirements and deployment setting. Even the administrators of the nodes where MAs are likely to execute could specify migration policies for incoming MAs, typically for load balancing purposes or for security management purposes.

Conflicts between policy specifications could arise due to omissions, errors or conflicting requirements of the users specifying the policies. Conflicts could also arise at run-time when one MA can be in the condition to apply different contrasting policies at event occurrence. This is the case of two migration policies, one defined by the application user and one by the agent system administrator, that command the same mobile agent to migrate at the same event occurrence, but toward two different nodes. To address this issue, the research already carried out in the field of policy-based management of distributed systems and requirement engineering can offer valuable solutions [12], [13], [14]. Currently, we face the problem of conflicts between policy specifications by relying on the solutions provided by the Ponder framework [12]. In the case of run-time conflicts, we adopt a default strategy: policies defined by system administrators have the precedence on user-defined migration policies.

When a new policy is enabled, i.e., correctly specified, analysed and compiled, it is ready to be stored in the Repository Service, and distributed to the interested entities, i.e., to policy subjects.

The *Repository Service* (RS) is organised as a distributed directory service that persistently stores policies. In particular, each directory entry records the Ponder policy specification along with the Java policy class representing it and generated by the Ponder policy compiler. In addition, each directory entry has a distinguished name that permits to uniquely refer to and retrieve its stored policy.

RS currently exploits the LDAP directory service provided by the SOMA infrastructure, and uses the SOMA naming system (also LDAP-compliant) to obtain the names and locations of policy subjects/targets [10]. The SOMA LDAP directory service also maintains references and attributes related to SOMA agents, places and domains.

Any change of policies and SOMA resources is represented in terms of an event that is generated through the Java Naming and Directory Interface event listener functionality provided by the LDAP server and notified to interested entities by the Event Service.

The *Distribution Service* (DS) is constantly informed about changes in the set of both policies and SOMA resources. Whenever a policy change occurs, as in the case of dis/enabled policies, the DS coordinates with the SOMA naming service to locate interested policy subjects and activates the un/loading of policies. In the case of variation in the set of SOMA resources, such as whenever a new agent is instantiated, the DS is responsible for retrieving from the RS the relevant policies and for initialising them into the agent.

We have decided to implement the DS in terms of a stationary *Distribution Agent* (DA), one for each SOMA domain. DAs are designed to autonomously perform and coordinate policy distribution among different domains (see Figure 2).

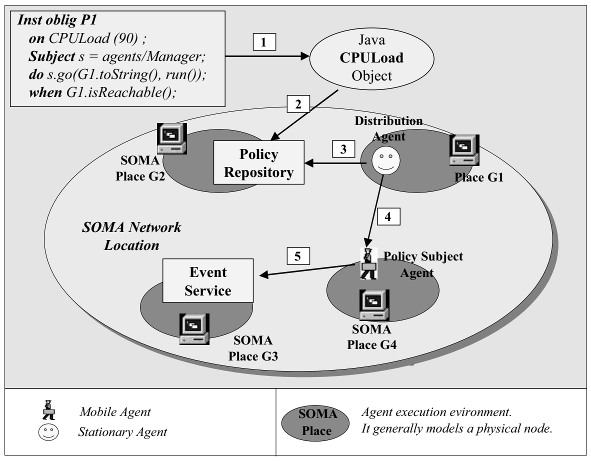
```
Inst oblig P1
  on CPULoad (90) ;
  Subject s = agents/Manager;
  do s.go(G1.toString(), run());
  when G1.isReachable();
```

Figure 2. Policy Initialization.

In particular, when a policy is first enabled, DA is responsible for extracting the correspondent Java policy object from the RS and of parsing it to retrieve relevant policy information: events, subject, target, actions, and constraints. Then, DA coordinates with the SOMA naming service to check whether policy subjects are active in the system and to retrieve their current location.

Finally, DA distributes policy objects to relevant policy subjects and registers policy events to the Event Service on behalf of policy subjects. Finally, note that the DA is currently implemented as a single stationary agent, but we are working to organise it as a set of cooperative agents to decentralise policy distribution management in the case of a very large number of policies.

3.2. Policy Enforcement

Policy enforcement is performed by exploiting the support facilities of the *Policy Enforcement Service*

(PES), as shown in Figure 3. We have implemented the PES as a set of coordinated specialised sub-components, each delegated to carry out a specific policy management task.

The *Policy Loader* is in charge of retrieving all migration/management Java policy objects given their GUID. It exploits the Java class loading mechanism for localising and dynamically loading policies. The *Policy Interpreter* parses and interprets all elements of Java policy objects. The *Constraint Verifier* extracts constraint specifications from Java policy objects, and verifies them by coordinating with the Monitoring Service for the evaluation of the current state of the execution environment. The *Policy Enforcer* is the core component that is delegated by SOMA agents to supervise policy enforcement tasks. In particular, it coordinates with all the other sub-components and executes on behalf of SOMA agents specified policy actions when the other sub-components have completed their tasks.

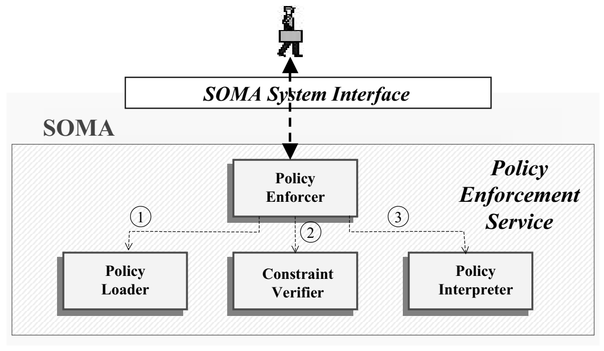

Figure 3. The Policy Enforcement Service.

It is worth outlining that the policy middleware ensures also the security of all policy lifecycle management tasks. Only authorised users can define migration policies and only authorised system entities can distribute and enact changes in the agent mobility behaviour at run-time. Secure policy distribution and enforcement is achieved by exploiting the wide set of security mechanisms available in the SOMA environment to provide the requested level of security [15].

3.3. Monitoring and Event Support

Policy activation and enforcement rely on the event monitoring and delivery provided by SOMA. The SOMA Event System (ES) currently integrates both monitoring and event registration/notification functions. Any SOMA domain provides an ES component that senses location/context modifications within the domain, possibly aggregates these modified indicators in higher level event objects, and notifies events to any entity subscribed for them. Events can be viewed as high-level objects that encapsulate information related to the changes they represent and that can be organised in extensible hierarchies to accommodate application requirements and environment characteristics.

The ES is currently designed to send events directly to subscribed SOMA agents even in case of migration. The current implementation of the dispatching functionality exploits the JEDI technology [16].

From the monitoring point of view, ES can observe the state of system and application resources, from the percentage of CPU usage on one place to the heap memory space allocated or to the network bandwidth consumed by the threads of a SOMA agent. ES integrates the Java Virtual Machine Profiler Interface to enable the observation of the JVM state at the application level, and platform-dependent monitoring modules via the Java Native Interface to obtain visibility of resource state at the kernel level.

4. Policy-controlled Mobile Agents

A SOMA agent is composed of three parts: state, code, and mobility policy. The SOMA agent code contains the static description of the computation to carry out. Migration and mobility management instructions are not included in the agent code. The mobility policy contains all Ponder obligation policies that have the agent as policy subject.

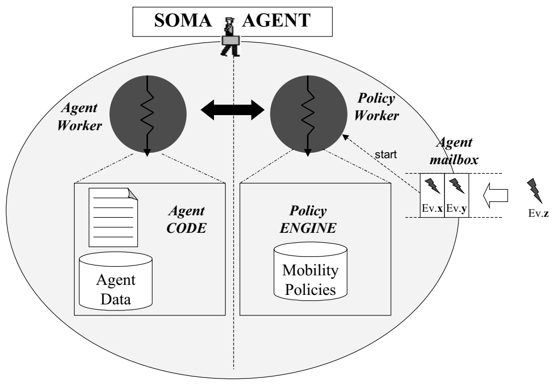

Figure 4. SOMA Agent Architecture.

When one agent is first instantiated, it is associated with a mailbox, with two active threads (the *Agent Worker* and the *Policy Worker*), and with a Policy Engine component (see Figure 4). The agent mailbox permits the agent to receive event notification. The Agent Worker thread is in charge of executing the agent code, whereas the Policy Worker thread is responsible for coordinating the activation of agent migration accordingly to mobility policy specifications. In particular, the Policy Worker is designed to remain in a sleeping state until an event notification is delivered to the agent via mailbox. At event notification the Policy Worker enters in a running state, delegates policy management operations to the Policy Engine component and waits until it terminates its tasks. Note that when the execution of a policy starts, any additional event notification is stored sequentially in the agent mailbox in a FIFO stack and it is extracted as soon as the management of the previously triggered policy terminates. The Policy Engine component provides the agent with all the functionality to load/unload policies into the agent mobility policy part and to activate policy retrieval and execution at event notification.

Note that the concurrent execution of both Agent Worker and Policy Worker threads may lead to inconsistencies, such as when the Policy Worker activates agent relocation while the Agent Worker is executing a critical section, i.e., a portion of agent code whose execution cannot be interrupted. Inconsistencies arise because the Java technology, used to implement SOMA agents, does not preserve the agent execution state upon migration.

Examples of critical sections are when the agent state contains results of partially completed operations, or when it is using local resources to which it cannot re-bind once migrated or when it is currently engaged in a transaction that has to be completed locally.

To avoid potential inconsistencies, agent migration should be delayed until the end of the critical section. Toward this goal we synchronise thread execution by providing agent programmers with the capability to mark a piece of code as a critical section. This informs the agent system to support mutual exclusive thread execution. For example, if the Policy Worker tries to activate policy enforcement while the Agent Worker is executing in a critical section, the Policy Worker computation is automatically suspended and resumed as soon as the execution of the critical section terminates.

4.1. Policy Mobility Types

In dynamic application scenarios a large number of mobility policies can apply to any SOMA agent to take into account the various changes that can occur in the agent execution environment. This number can also increase during agent life-cycle to follow the evolution of application requirements and environment state.

SOMA provides several strategies for policy distribution that can be selected dynamically depending on application requirements. At one extreme it is possible to load all policy Java objects directly into an agent. This favours the prompt reaction of the agent to events, but increases the agent size and can hinder the efficiency of agent migration. At the other extreme, the agent is not loaded with migration policies that are instead retrieved on demand from the RS at event occurrence. The programmer may also choose any intermediate solution to fit application requirements. The choice of the policy distribution strategy should consider the characteristics of the different types of policies. Figure 5 shows the taxonomy of the mobility policy types currently supported in SOMA. The mobility policies that rule SOMA agent mobility behavior are divided into two main subsets: the *System Policy* and the *Agent Policy* set.

The former set contains policies that are used to define reactive mobility and that are written by system administrators typically for load balancing purposes. The policy *P2* reported in Table 1 is an example of System Policy.

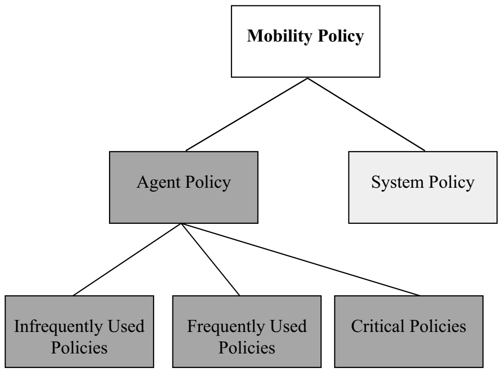

Figure 5. Mobility Policy Taxonomy.

The *Agent Policy* set contains policies for supporting proactive mobility (see policy *P1* in Table 1). This means that these policies are the ones that the DAs have to distribute and to load into SOMA agents. To tradeoff between the need of prompt agent reaction to event notification and efficient agent migration, we have decided to decompose Agent Policies into the following subsets: *Infrequently Used Policies*, *Frequently Used Policies* and *Critical Policies*.

Infrequently Used Policies define migration decisions that depend on circumstances that are expected to occur infrequently during agent lifetime. An example is a policy that forces a mobile agent to return back home after a predefined number of migration hops. Because of potential infrequent activation, these types of Java policy objects are stored only in the RS, whereas just their policy references, i.e., their GUIDs, are loaded into the agent. At policy activation time, the agent will retrieve the triggered policy object on demand by exploiting the Policy Loader support.

Frequently Used Policies bind agent migration to events that occur very often in any agent execution environment. In this case, the Java policy objects are stored directly into the agent to avoid frequent connections with the RS.

Critical Policies are used to control agent migration when critical situations occur that can compromise the survivability of agent execution. Critical policies can be typically exploited to leverage application fault-tolerance. For instance, a critical policy could oblige an agent executing on a mobile device to migrate when the battery power of the device decreases under the minimal threshold. Similarly to the case of Frequently Used Policies, Java policy objects related to Critical Policies are stored directly into the agent.

5. Case-Study

Network and systems management proposes a relevant arena for demonstrating the benefits of using the MA technology [17], [18]. In this context we have tested the validity of our policy approach in achieving separation between mobility and computational concerns and the usability of our policy infrastructure. In particular, we have built a MA-based management system targeted at supporting load balancing over a set of network nodes.

The idea is to provide system and network administrators with policy-driven MAs that can move among managed nodes and perform locally load balancing management tasks on the behalf of administrators. The underlying assumption is that the set of nodes to manage is not fixed, but can vary dynamically due to the addition/deletion of new nodes.

As far as load balancing is concerned, MAs control at each node that some specific processes do not exceed their assigned resource consumption threshold. In the case of excessive resource consumption, MAs may suspend or kill the responsible process or move it to a less loaded node. With regard to network exploration, MAs can adopt various migration strategies. One possibility for MAs could be to roam the set of nodes sequentially in a Round Robin fashion. This strategy assumes that all nodes have the same management priority. In particular, when a node connects to the network, its name is appended to the list of nodes to visit. Alternatively, MAs could assign a priority to each node depending on its load and visit first the most overburden nodes.

Figure 6 compares the conventional programming approach to agent mobility with our policy-controlled one.

Traditionally Programmed MA: **(a)**

```
public class BalancerAgent extends Agent{
 LinkedList ItineraryPath ;
 Hashtable load = new Hashtable();
 PlaceID currentPlace;   int average; int currentLoad;

 void run() {   // Initialisation method
   ...      // ItineraryPath initialisation
 mailbox.mailListener = new Mailbox.MailListener(){
   public void runListener(){
     while (mailbox.isMessage()) {
       Message mess = mailbox.getMessage();
       if (mess instanceof newNodeMessage){
         PlaceID newPlace = (PlaceID) mess.message;
         if ( newPlace.QoSLevel == "High")
              ItineraryPath.addFirst(newPlace);
         else ItineraryPath.addLast(newPlace); }
   }}};
 startMethod(); }
// re-starting method after migration
void startMethod() {
currentPlace = super.getPlaceID();
retrieveLocalLoad();
average = dataElaboration();
balancingActions();
go();  }

void go() {
PlaceID node;
 if (ItineraryPath.size()==0) return;
 try{
 if ((currentLoad  >  100) {
     int middle = ItineraryPath.size() /2;
     ItineraryPath.add(middle, currentNode);
 }else {     // Round-Robin
     ItineraryPath.addLast(currentNode); }
 // migration to the next node
 node = (PlaceID) ItineraryPath.removeFirst();
 super.go (node, "startMethod");
 }catch(MigrationException e){ ...}      }
void retrieveLocalLoad(){
currentLoad = checkLocalLoad();
load.put(currentPlace, currentLoad);}

void balancingActions() {... }
int checkLocalLoad{ ... }
int dataElaboration() { ... }
...  } // BalancerAgent End
```

Policy Controlled MA: **(b)**

```
public class BalancerAgent2  extends PolicyAgent{
  LinkedList ItineraryPath ;
  Hashtable load = new Hashtable();
  PlaceID currentPlace; int average;  int currentLoad;

void run() {   // Initialisation method
   .....    // ItineraryPath initialisation
  startMethod(); }
void startMethod() {
   currentPlace = super.getPlaceID();
   criticalSection("retrieveLocalLoad");
   average = dataElaboration();
   criticalSection( "balancingActions" );  }
 void retrieveLocalLoad() {
    currentLoad = checkLocalLoad();
    load.put(currentPlace, currentLoad); }
int balancingActions(){ ... }
int dataElaboration(){ ... }
int checkLocalLoad{ ... }
} // BalancerAgent2 End
```

(c)

```
inst oblig MobPol1
on NewConnection( node );
subject s= res/agents/BalancerAgent2;
do s.ItineraryPath.addFirst(node) ;
when node.QoSLevel = "High";
```

```
inst oblig MobPol2
on TaskCompletion ( BalancerAgent2);
subject s= res/agents/BalancerAgent2;
do s.ItineraryPath.addLast(s.currentNode)
    -> node = s.ItineraryPath.removeFirst()
    -> s.go (node, "startMethod");
when ItineraryPath.size()>0 and s.currentLoad < 100
```

```
inst oblig MobPol3
on  TaskCompletion ( BalancerAgent2);
subject s= res/agents/BalancerAgent2;
do middle = ItineraryPath.size() /2
 s.ItineraryPath.add(middle, s.currentNode)
 -> node = s.ItineraryPath.removeFirst()
 -> s.go (node, "startMethod");
when ItineraryPath.size()>0 and s.currentLoad > 100
```

```
inst oblig MobPol4
on  SecurityAlarm( node );
subject s=
res/agents/BalancerAgent2;
do  s.ItineraryPath.remove(node);
```

Figure 6. The comparison of a traditional SOMA agent with a Ponder-controlled one.

In particular, in the traditional approach the code embeds both load balancing and migration directives (Figure 6a). The agent, called *BalancerAgent*, extends the abstract *Agent* class that defines the basic methods for initialising agent execution (*run()* method), for specifying agent tasks (*startMehod()* method), for supporting agent migration (*go()* method), and for allowing agents to retrieve status information related to their current execution environment (*getPlaceID()* method).

The *run()* method initialises the initial list of nodes to visit (*ItineraryPath* variable) and the local mailbox (*Mailbox* variable) for supporting agent communication with external system entities. A mailbox listener is defined to handle incoming messages. When a new message is delivered in the agent mailbox, the *runListener()* method is responsible for reading and processing the message content. In the example, if the message *mess* notifies that a new node has connected to the network (message of kind *newNodeMessage*) then the node name is inserted in the *ItineraryPath* list depending on its level of negotiated QoS. If, for instance, the QoS level to be granted is high, the node name is added to the head of the itinerary list. The *startMethod*() method specifies the agent tasks to execute at each node. In particular, the agent checks the resource consumption of local processes and stores this value in a local Java Hashtable (*retrieveLocalLoad()* method). Then it re-calculates the new average node load (*dataElaboration()* method) and performs load balancing actions (*balancingActions()* method).

Finally, at load balancing task completion, the *BalancerAgent* selects the next node to visit and migrate toward it (*go()* method). In the example, for sake of simplicity, we assume that the MA migrates always toward the first node of *ItineraryPath* list and adapts its migration strategy by changing the order of nodes in the *ItineraryPath* list. In particular, the MA exploits a Round-Robin migration strategy if the nodes are not highly overburden. Otherwise, in the case the load of a node exceeds a pre-defined threshold, e.g. 100, the MA gives a temporary priority to the node by adding it to the middle of the *ItineraryPath* list instead of appending it to the tail. Therefore, the time between two exploration of an overburden node is reduced to half with respect to the usual Round-Robin strategy.

In the policy-based approach only the application logic targeted at supporting load balancing is embedded into the agent code part, whereas the agent mobility behaviour is abstracted away and modelled by means of Ponder obligation policies. In particular, the policy-controlled SOMA agent called *BalancerAgent2* extends the abstract *PolicyAgent* class that provides agents with the methods to load/unload policies and to react to event occurrence accordingly to mobility policy specifications. Agent code computation consists in the execution of only the methods required for performing load balancing.

The set of Ponder policies that rule agent mobility are shown in Figure 6c. Whenever a new node joins the network, the *MobPol1* policy forces the *BalancerAgent2* agent to add the new node to the head of the *ItineraryPath* list. The policy is triggered by the *NewConnection* event that notifies the connection of a new node to the network (the *on* clause) and can be applied only if the level of negotiated QoS for the new node is high (the *when* field). *MobPol1* is an example of Infrequently Used Agent Policy. This derives from our assumption that new nodes can join the network, but this is unlikely to occur so frequently.

When the agent completes its tasks (*TaskCompletion* event), it can exploit two alternative migration strategies. When the node load is under the specific threshold, the *MobPol2* triggers the Round Robin migration strategy. On the contrary, *MobPol3* forces the agent to add the current node to the middle of *ItineraryPath* list. Being activated each time the agent achieves its management goals, both *MobPol2* and *MobPol3* policies can be considered examples of Frequently Used Policies.

Let us show how this set of policies can effectively rule at runtime the agent mobility behaviour by focusing on *MobPol1*. When the SOMA Event Service detects a new node connection, it delivers the *NewConnection* event to the agent. At event reception, the Policy Worker thread of the *BalancerAgent2* agent has to retrieve the Java policy object associated to the notified event in order to trigger its enforcement. Because the agent owns only a reference to the policy object (*MobPol1* is an Infrequently Used Policy) the Policy Worker delegates to the PES policy retrieval, in addition to policy interpretation and enforcement. In particular, the Policy Loader first retrieves the policy object from the Repository Service and passes it to the Policy Enforcer that coordinates with the other PES components to check policy constraints, to interpret policy information and to enforce migration accordingly to policy specifications. Once the PES has completed policy enforcement, the control returns to the Policy Worker that can either wait for another event message (if the agent mailbox is empty) or pop from the FIFO stack an event message if available.

Similar considerations apply for the enforcement of *MobPol2*. There are only few differences. The primary one is that the agent owns the Java policy object because *MobPol2* is a Frequently Used Policy. Therefore, the Policy Worker, via the Policy Engine, extracts the policy object directly from the agent state and passes it to the PES for its interpretation and execution. Another difference is that if the policy constraints are not verified, the Policy Worker tries to enforce the *MobPol3* that is triggered by the same event.

It is worth noting that the execution of the Agent Worker is synchronised with the Policy Worker. The fragment of application logic code that can cause a wrong load balancing if abruptly interrupted, is the one in which the agent either reads and stores the *currentLoad* of the node or performs load balancing tasks. For instance, an inconsistent read or storage of the *currentLoad* can cause a wrong estimation of the average and consequently the execution of erroneous load balancing actions. The agent programmer can use the *criticalSection()* method of the *PolicyAgent* class to mark the *retrieveLocalLoad()* and *balancingActions()* methods that contain these fragments of code as critical. The method causes the caller thread to acquire a software lock. This ensures that at event notification, the Policy Worker cannot delegate to the PES policy enforcement until the lock is released.

As a final remark, policies can vary during the agent lifetime due to changing administrative preferences. In the case of changes, administrators can disable old policies and substitute them with new ones. Suppose, for instance, that *MobPol4* is required at run time to avoid agent migration to an unsafe node. In particular, if a managed node has been attacked by intruders (as notified by the *SecurityAlarm* event), the *MobPol4* policy forces the agent to temporarily remove the name of node from the *ItineraryPath* list. *MobPol4* is an example of Critical Policy.

Note that in the traditional programming approach to mobility the addition of *MobPol4* requires to change the agent code part to take into account this new situation. On the contrary, in our policy-controlled approach, administrators have to simply define the new policy without having to modify the agent code. It is the underlying policy infrastructure that transparently enables the new policy and propagates it to interested agents.

6. Related Work

This section focuses on the few proposals that, to the best of our knowledge, explicitly deal with dynamic mobility management in applications built with mobile code technologies. All solutions have in common the principle of separation between mobility and computational concerns as the key design requirement. However, the proposals differ on how to achieve this separation of concerns.

The approach described in [7] proposes to separate an MA application into three aspects, the function, the mobility and the management aspects, and to program them separately. Each MA is associated with an array with three key/value pairs: a code that gives the code of the agent, a data that contains data and the MA references to needed resources, and a path that gives the itinerary of the agent.

Another interesting approach is the **FarGo** system that allows to program the layout of a mobile application separately from the basic logic [8]. Dynamic layout policies are encoded within the application by using special API. The use of an event-based scripting language for the specification of layout policies externally to the application code is also considered and is currently under development.

Another recent proposal is represented by the **MAGE** model that introduces the programming abstraction of mobility attributes to describe the mobility semantics of application components [9]. Programmers can attach mobility attributes to application components to dynamically control the placement of these components within the network. However, MAGE relies on the programmer to manually enforce the binding between a program component and its mobility attributes. In addition, MAGE does not currently integrate any security model that controls both the specification and run-time enforcement of mobility attributes.

In [19] another solution is proposed to support incremental insertion or substitution of, possibly small, code fragments at run-time. Differently from the previous proposals that are Java based, **XMILE** exploits XML to achieve more fine-grained mobility than in the approaches based on Java. XMILE enables complete programs as well as individual lines of code to be sent across the network. XMILE is more a model rather than a ready-to-use framework for dynamic application reconfiguration.

Another significant proposal is the **DACIA** framework that provides support for the construction and execution of reconfigurable mobile applications [20]. DACIA provides mechanisms to change the application structure at runtime, but no high-level languages are integrated to support the specification of reconfiguration policies clearly separated from the application code.

Our proposal has several points in common with the described approaches; the main difference is in the possibility to specify mobility strategies at a higher level of abstraction and to modify them even during the application execution. In addition, our policy infrastructure with its wide set of support services can be considered a useful ready-to-use environment for the design, development and support of adaptive MA applications in a wide variety of application scenarios.

7. Conclusions

Most MA systems exhibit rigid agent migration control schemes. The directives for governing agent mobility are typically hard-coded and tangled within the agent application logic. The lack of separation of concerns forces mobile application designers to take into account at design time both algorithmic and allocation issues and to re-implement agent code in the case migration strategies need to be changed to accommodate evolving conditions.

Only recently few proposals have started to emerge to provide more flexible approaches to mobile application layout reconfiguration. Along this direction, we propose a policy-based approach to mobility and a policy framework for the dynamic control of agent mobility behaviour.

The framework results from the integration of a policy-based management system, called Ponder, in the SOMA mobile agent environment. Within this framework, programmers specify agent mobility behaviour in terms of declarative Ponder policies. Dynamic modification of agent mobility behaviour requires to change only policy specifications, whereas the control and enforcement of migration accordingly to policy specifications is delegated to the underlying policy middleware, transparently to agent programmers.

Experiences in the use of the proposed policy-based mobility model have shown that our middleware can simplify the design and implementation of MA-based services in a wide variety of different usage scenarios. These encouraging results are stimulating further research along different guidelines to improve the current prototype and to develop more complex services on top of it. In particular, we are working on extending the middleware to support flexible mobility management even in the case of strict resource-limited portable devices with intermittent connectivity.

As a final remark, we are convinced that our policy-based approach could extend to other application problems not strictly related to agent mobility. Areas that are currently under investigation are binding management upon agent migration and physical mobility. We are convinced that all results shown in this paper could be easily ported to these application domains.

References

[1] A. Fuggetta, G.P. Picco, and G. Vigna, "Understanding Code Mobility", *IEEE Transactions on Software Engineering*, Vol. 24, No. 5, May 1998.

[2] G. Picco, "Mobile Agents: An Introduction", *Journal of Microprocessors and Microsystems*, Vol. 25, No. 2, April 2001.

[3] R. Wies, "Using a Classification of Management Policies for Policy Specification and Policy Transformation", *Proc. of ISINM'95*, Chapman & Hall, 1995.

[4] N. Damianou, et al., "The Ponder Policy Specification Language", *Proc. of Policy '01*, Springer Verlag, LNCS 1995, Bristol, UK, 2001.

[5] A. Corradi, et al., "Policy Controlled Mobility", *ICSE'01 Workshop on Software Engineering and Mobility*, Toronto, Canada, 2001.

[6] G.P. Picco, "μCode: A Lightweight and Flexible Mobile Code Toolkit", *Proc. of MA'98*, Springer Verlag, Stuttgart, Germany, 1998.

[7] K. Lauvset, et al., "Factoring Mobile Agents", *Proc. of ECBS'02 Workshop*, Lund, Swe-den, IEEE CS Press, 2002.

[8] O. Holder, et al., "Dynamic Layout of Distributed Applications in FarGo", *Proc. of ICSE'99*, ACM Press, Los Angeles, California (USA), 1999.

[9] E. Barr et al., "MAGE: a Distributed Programming Model", *Proc. of ICDCS'01*, IEEE Press, Phoenix, Arizona (USA), 2001.

[10] P. Bellavista, et al., "A Secure and Open Mobile Agent Programming Environment", *Proc. ISADS '99*, IEEE Press, Tokyo, Japan, 1999.

[11] N. Damianou, et al., "Tools for Domain-based Policy Management of Dystributed Systems", *Proc. of NOMS'02*, IEEE Press, Florence, April, 2002.

[12] E. Lupu, et al. "Conflicts in Policy-based Distributed Systems Management", *IEEE Transactions on Software Engineering*, Vol. 25, No. 6, 1999.

[13] J. Chomicki, et al., "A Logic Programming Approach to Conflict Resolution in Policy Management", *Proc. KR2000*, Morgan Kaufmann Publishers, Breckenridge, Colorado (USA), 2000.

[14] W. N. Robinson et al., "Managing requirements inconsistency with development goal monitors", *IEEE Transactions on Software Engineering*, Vol. 25, No. 6, 1999.

[15] A. Corradi et al., "Security of Mobile Agents in the Internet", *Internet Research*, Vol. 11, No. 1, MCB University Press, 2001.

[16] G. Cugola, et al., "The JEDI Event-Based Infrastructure and Its Application to the Development of the OPSS WFMS", *IEEE Transactions on Software Engineering*, Vol. 27, No. 9, September 2001.

[17] M. Baldi, G. P. Picco, "Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications", *IEEE Int. Conf. on Software Engineering (ICSE'98)*, April 1998.

[18] B. Pagurek, Y. Wang, T. White, "Integration of Mobile Agents with SNMP: Why and How", *IEEE/IFIP Network Operations and Management Symposium (NOMS'00)*, USA, April 2000.

[19] C. Mascolo et al., "XMILE: an XML based Approach for Incremental Code Mobility and Update", *Automated Software Engineering Journal (Special Issue on Mobility)*, Vol. 9, No. 2, Kluwer Publisher, April 2002.

[20] R. Litiu, "Providing Flexibility in Distributed Applications Using a Mobile Component Framework", *Ph.D. dissertation*, University of Michigan, Electrical Engineering and Computer Science, September 2000.