How to Support Adaptive Mobile Applications

Antonio Corradi, Rebecca Montanari,
Gianluca Tonti
Dip. di Elettronica, Informatica e Sistemistica
Università di Bologna
Viale Risorgimento 2, 40136 Bologna, Italy
{acorradi, rmontanari, gtonti}
@deis.unibo.it

Cesare Stefanelli
Dipartimento di Ingegneria
Università di Ferrara
Via Saragat 1, 44100 Ferrara, Italy
cstefanelli@ing.unife.it

Abstract

The Mobile Agent (MA) paradigm seems to be a promising solution for the design and development of distributed value-added services. However, mobility has added complexity to the design of application and requires new programming methodologies to allow agents to adapt their migration strategies to the current context and to react to unforeseen events. In this paper, we present a policy-based approach to mobility that allows application designers to express and govern mobility decisions at a high level of abstraction and to dynamically change the allocation of agents across the network nodes without intervention on the agent code. Policies are exploited to specify mobility strategies and allow to abstract away the specification of migration choices from the component code. The paper proposes a layered policy-aware architecture for MAs that allows to adapt the mobility behaviour of agents at different levels of abstraction depending on programmer expertise. Unskilled programmers can specify simply when and where to migrate agents, whereas skilled programmers can also specify which migration mechanism to select and activate to better accommodate to environment conditions. We have experienced the dynamicity and flexibility of the proposed approach in the framework obtained by integrating a policy-based management system in a mobile agent environment.

1. Introduction

The rapid growth of the Internet and the widespread popularity of the Web together with the increase of user requirements and expectations for value-added services have motivated the interest in new and flexible programming paradigms based on mobile entities. A significant example is represented by the Mobile Agent (MA) model that proposes the mobility to a remote site of a whole computational component along with its state, the code it needs, and some resources required to perform the task [1].

The MA technology can offer promising solutions for the design of distributed advanced services. Mobile agents execute locally to the information servers thus providing better network utilization and better support to mobile users, which can disconnect while their agents roam in the network and autonomously execute tasks on their behalf. Mobile computing, electronic commerce, network management, and distributed information retrieval are typical application scenarios that have started to show the benefits deriving from the adoption of MAs [2], [3].

However, mobility has added complexity to the design and deployment of applications in wide area computing environments. In the MA paradigm the abstraction of location is explicitly introduced becoming a first design concept. Programmers have to explicitly handle agent allocation onto the set of available environment nodes. This can be challenging because it requires application designers to express and decide when and where to move executing agents and to have a deep knowledge of application and environment details to achieve a proper allocation of agents. However, even a high programming expertise is not sufficient. Programmers still have to cope with the complexity deriving from the need to modify at run time the layout of MA applications to reflect the dynamicity of the environment where agents tend to execute. In environments that cannot assume fixed logical and physical network boundaries, migration strategies cannot derive from a-priori assumptions on which resources and nodes are available at agent execution time and cannot be statically hardcoded into agent codes. The traditional programming approach of migration rules built

Investigation supported by the Italian “Consiglio Nazionale delle Ricerche” in the framework of the Project “Global Applications in the Internet Area: Models and Programming Environments” and by the University of Bologna (Funds for Selected Research Topics: "An integrated Infrastructure to support Secure Services").

into the agent code lacks flexibility and involves extensive re-engineering efforts at every unforeseen change.

We claim that mobility concerns should not be mixed with the agent application code, but rather clearly separated in order to promote dynamic adaptation of agent migration strategies to the evolving application requirements and to the dynamic changes of the execution context. New programming paradigms are required that allow programmers to dynamically change the allocation of agents without any impact on their code and to handle mobility at a higher level of abstraction to hide system and network specifics.

Toward this goal, we claim that a policy-based approach can provide a promising solution to achieve adaptive agent mobility behaviour [4]. Policies are rules governing choices in the behaviour of a system separated from the components in charge of their interpretation [5]. In the context of mobility, policies can be specialised to specify choices in the mobility behaviour of agents in terms of when and where agents have to migrate. The clear separation between migration rules and agent code permits to change agent mobility behaviour without re-implementing agents themselves.

An additional advantage is that policies allow programmers to uniformly express adaptation requirements that typically arise in a mobile scenario at different levels of abstraction. Policies can be used not only to adapt agent migration in terms of when and where to migrate to accommodate both application requirements and dynamic environment conditions, but also to dynamically select the most appropriate agent migration mechanisms to tailor to the current environment state. Changes in network connections or bandwidth availability can, in fact, affect how agents migrate toward destination nodes. Neglecting these variations could lead to a degradation of the overall quality of service offered to users, while taking them into account permits to deliver a high level quality of service.

This paper describes a MA system architecture that integrates and exploits a policy-based mobility model for dynamically adapting the mobility behaviour of agents. Mobility strategies are expressed in terms of declarative high-level policies decoupled from the application logic and can be modified by simply changing policy specifications. The feasibility of the approach is demonstrated by the integration of the Ponder policy language within the Java-based SOMA mobile agent [6], [7]. The Ponder language defines and controls agent mobility strategies and is exploited to specify mobility policies at both the application and the middleware level, while the SOMA programming framework offers the needed services for the support of policy-driven mobile agents applications.

2. Policy-driven Adaptation of Agent Mobility

2.1. Design Requirements

The adoption of a policy-based approach to dynamically adapt the mobility behaviour of agents involves the definition of a high level policy language to facilitate the specification of mobility requirements and the development of an efficient and effective run-time support for policy definition enforcement.

We propose to adopt declarative policy languages to simplify policy specification tasks and to favour mobility policy analysis and verification. In particular, mobility decisions can be expressed in terms of declarative *event-triggered* policies, i.e. policies triggered by event occurrences. Event-triggered policies specify the migration actions that must be performed when specific events occur. Events can be defined as generic state changes in the system and can uniformly model different changes in application and environment conditions. Some examples of the former case are the completion of an application task and the arrival/departure of agents in/from an execution context. For the latter case, consider the CPU and the memory usage, and the bandwidth occupation.

We have identified other properties that the design of a policy language for mobility should guarantee. A policy language for mobility has to support explicit constructs to permit the definition of all the basic elements involved in mobility. First of all, the language should enable the specification of the entity that is in charge of triggering the mobility strategy, i.e. the policy subject, and the entity which is targeted by relocation, i.e. the policy target. A policy subject could be either a mobile agent or the current execution environment. In the first case, the mobile agent autonomously decides to migrate, whereas in the second case it is the hosting execution environment that forces the migration, for instance to improve load balancing. In addition, because we adopt event-triggered polices, the policy language should be designed to model the sequence of events relevant for triggering migration changes, the set of mobility actions to perform at event occurrence, and the location where mobile components need to move. In addition, language constructs should be available to enable us to explicitly define the conditions to be checked before the actual activation of mobility actions. These conditions act as constraints for the applicability of a mobility policy and can depend on dynamic attributes, such as the state of the resources that the agent is accessing. For instance, a programmer could be willing to block agent migration if the destination host is unreachable.

Another essential requirement is to design a platform-independent policy language for mobility that can map to and co-exist with one or more existing underlying mobility-aware platforms.

The run-time policy support is responsible for translating platform-independent policy specifications into low-level implementable policies and for managing policy lifecycle, from policy initial distribution to policy activation and control. In particular, the run-time policy support has the essential goal of ensuring a consistent and secure policy activation. Automatic and transparent policy lifecycle management requires a set of support services. We have identified the following needed facilities [8]:

- a *specification service* for supporting the editing/browsing of policies, for policy analysis and verification and for the mapping of high level policy specifications into low-level platform-dependent rules;
- a *distribution service* for policy loading/unloading from the entities to which policies apply;
- an *enforcement service* for interpreting at run-time low-level event-triggered mobility rules at any event occurrence. Because the policy-based mobility model relies on event-triggered policies, the policy enforcement service needs to rely on *event services* for the registration, detection and notification of events to entities interested in their occurrences.

2.2. A Layered Architecture

High-level policy languages and correspondent policy run-time supports are primary building blocks that need to be effectively and consistently integrated within a MA system to obtain an adaptive mobility architecture.

Figure 1 shows our proposal of a logical organisation of MA systems that allows application designers to dynamically tune agent mobility according to their application needs and environment conditions.

The logical architecture distinguishes system components and profiles. System components include agents at the application layer while, at the middleware layer, policy run-time support services along with the basic support mechanisms for agent execution, such as migration, communication, naming and event services.

Mobility adaptation is based on the profiles that we associate to agents and to the migration service. Profiles guide agent mobility in terms of when, where and how to migrate and, for this purpose, they contain a set of data and metadata.

Data allow to export awareness of the behaviour of both agents and execution environment to applications and simplify the inspection from outside of the characteristics of the components that are relevant for mobility choices. In particular, the data part related to agent profiles describes in a structured manner the current state and attributes of agents, such as user preferences and security settings. The data part of the migration service contains the description of its properties, in terms for instance of supported migration mechanism interface, along with information related to the current state of the execution environment (CPU load, bandwidth availability, ..).

Metadata are used to define and govern the behaviour of components depending on data content. In particular, metadata contain event-triggered policies, which define how changes in the execution context where agents execute should affect mobility decisions and actions. Metadata distinguish what is the desired mobility behaviour of components from how it is achieved [9]. Mobility decisions can be dynamically changed by simply adding/deleting policy specifications from the metadata field.

Metadata contain policies at different levels of abstraction to achieve different adaptation requirements. At the application layer, the policies in the metadata part of agent profiles specify only when and where agents have to move and not how. This permits programmers to neglect the details of agent migration mechanisms by simply specifying choices in the mobility behaviour of their agents according to their application preferences.

The selection of the migration mechanism that is most appropriate to accommodate current environment conditions is performed at the middleware level. This is achieved by means of the event-triggered policies contained in the metadata part of the migration service. Policies at this level are likely to be either specified directly by skilled programmers or made available by system administrators to provide a high level quality of service.

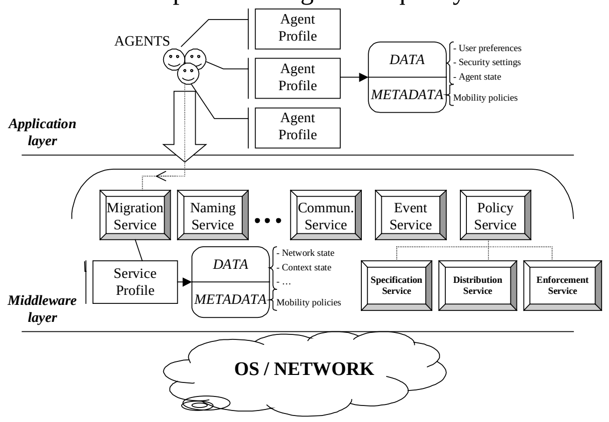


Figure 1. a layered architecture for supporting adaptive agent mobility**.**

3. A policy-based MA System

3.1. Ponder Policies for SOMA Agents

We have designed and applied the policy-controlled approach to mobility in the context of an MA system that exploits a policy language for governing agent behaviour. The MA system is called SOMA and it is a programming framework with a rich set of services for the development of secure and interoperable MA applications [7]. SOMA agents are controlled via policies expressed in the Ponder language.

Ponder is a declarative object-oriented language that satisfies the design requirements depicted in section 2.1. In particular, Ponder is designed to support the specification of several types of management policies for distributed systems and to provide structuring techniques for policies to cater for the complexity of policy administration in large enterprise information systems [6]. The policy-based mobility model only uses a subset of Ponder policies, i.e. obligation policies, which are event-triggered policies defining the actions that policy subjects must perform on target objects when specific events occur.

Let us introduce a simple example to illustrate Ponder obligation policies. In Table 1 the P1 policy states that the agent called *Manager* is obliged to migrate to a different node, called G1, when the current node becomes overloaded. The migration action is triggered by a CPU usage exceeding 90%, thus forcing the *Manager* agent to move to G1 and to perform the *run()* method there, if G1 is reachable.

Note that the triggering event specification follows the `on` keyword and that it can include events of any type, such as excessive CPU usage, by simply defining the event name and parameters. Event expressions can also be used to combine basic events into more complex ones, to allow to model any sequence of events. Multiple actions can follow the `do` keyword. In this case, the policy obligation language provides several concurrency operators to specify whether actions should be executed sequentially or in parallel. For instance, t.a1 -> s.a2 means that the action a2 on the object s must follow the action a1 on the object t. In addition, Ponder obligations can include a `when` clause to let programmers specify the set of conditions for the policy to be considered valid, e.g., the reachability of the G1 node in p1. These conditions must be explicit in the policy specification and act as constraints for the applicability of a policy. Ponder constraints are expressed in terms of a predicate, which must be true for the policy to apply.

```
 inst oblig P1
 on Load (CPU, 90) ;
 subject s = agents/Manager;
 do s.go(G1.toString, run());
when G1.isReachable();
```

Table 1. Ponder obligation policies.

For further details on Ponder and on the integration between Ponder and SOMA please refer to [8].

3.2. Case-Study

Let us consider a simple case study to show some advantages of our policy-based approach to mobility. Inside a multinational company network a mobile agent (called *Configurer* agent) is programmed for configuring and updating the common software that is installed in each machine of the network. In particular, this network consists of some domains, such as the headquarter LAN and the subsidiaries LANs (see Figure 2). One machine inside each domain is used as gateway to the external world and as local server for the other machines that are used as workstations. The number, type and location of the machines connected to each domain can dynamically vary and are autonomously managed by a local administrator. Therefore, the headquarter only owns the knowledge of the gateways of the company network, and not that of the local distribution of the machines within each domain.

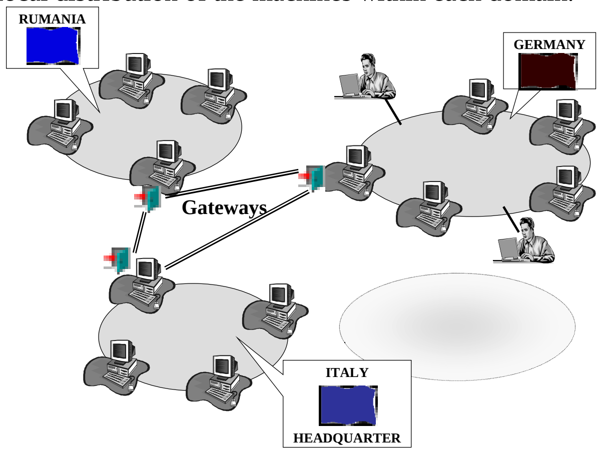
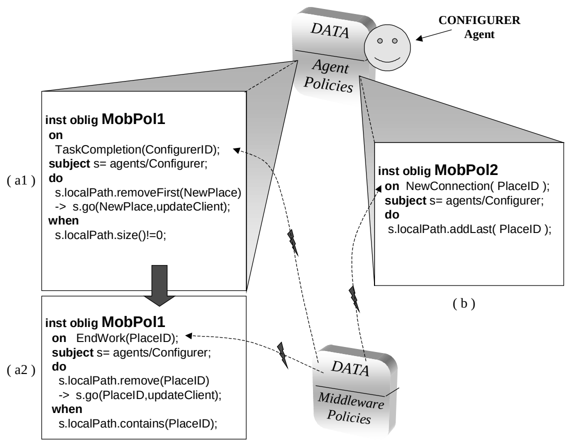


Figure 2. Case study**.**

SOMA system offers an easy way to model a network configuration like the one of our case study, by defining several abstractions for physical resources. The Place abstraction, where agent can execute, represents the physical machine. The Domain abstraction encloses a set of places and it typically represents a LAN. Each Domain includes a gateway, called Default Place, that is in charge of providing interconnections between different domains.
Inside this environment, the programmer of the *Configurer* agent has the capability to rule agent mobility through a set of Ponder policies. `MobPol1` policy (see Figure 3), is an example of how a programmer can rule both the moment (when) and the destination Place (where) for agent migration. This policy is triggered by the *TaskCompletion* event. This event notifies the end of the agent configuration task in a remote Place. The *Configurer* agent has to check for the next Place to visit in its *localPath* (method *removeFirst()* in the `do` field). The agent variable *localPath* contains the local set of machines to configure and is updated at runtime by the agent, by reading the gateway Data set. The policy can be applied only if there are other Places in *localPath* to configure (the `when` field).

The policy `MobPol2` is specified to adapt the set of Places to configure (where) at runtime, by adding new Places when they connect themselves to the domain. The triggering event for this policy is the notification of a new connection in the domain (*NewConnection* event), while the triggered action is the addition of a new connected Place to the end of the *localPath* list (*addLast()* method). This policy does not contain constraints, therefore it is always enforced after the notification of a *NewConnection* event.



```
inst oblig MobPol1
 on
  TaskCompletion(ConfigurerID);
 subject s= agents/Configurer;
 do
  s.localPath.removeFirst(NewPlace)
  ->  s.go(NewPlace,updateClient);
 when
  s.localPath.size()!=0;
```

```
inst oblig MobPol1
  on   EndWork(PlaceID);
  subject s= agents/Configurer;
  do
   s.localPath.remove(PlaceID)
   ->  s.go(PlaceID,updateClient);
  when
   s.localPath.contains(PlaceID);
```

```
inst oblig MobPol2
 on  NewConnection( PlaceID );
 subject s= agents/Configurer;
 do
  s.localPath.addLast( PlaceID );
```

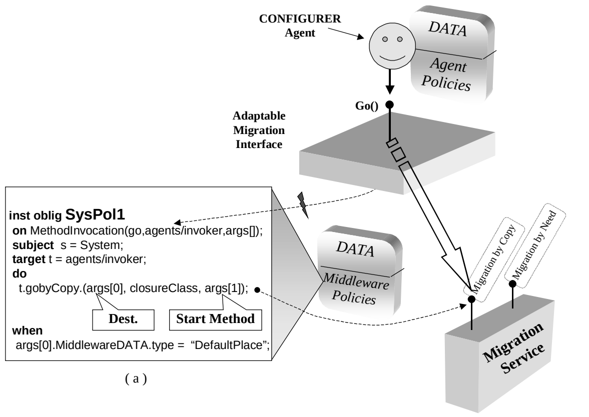
Figure 3. An example of Agent level policies.

The choice of how to migrate is delegated to the middleware level in a transparent way. In our case study, the middleware policies are common to all the domains and written by the unique administrator of the multinational company, but it is possible to have different policies for different domains. Unlike the higher level case, these Ponder policies are activated only by the events correlated and triggered by the call of the SOMA migration method (the *go()* method).

The SOMA migration service provides two agent migration mechanisms, *by need* and *by copy*, which allow the shipping of agent classes only when needed or in one shot at the beginning respectively. In our example, the default migration mechanism is a migration by need. However, it is useful to support a migration by copy when an agent wants to migrate from a gateway to another one. In this way, during its configuration task, the agent can load (by need) all the necessary classes from the gateway inside its current LAN, thus avoiding a frequent class loading from a remote domain.

To perform this goal, the administrator writes the policy `SysPol1` (see Figure 4) at Middleware level. When the Configurer agent calls the *go()* method (*MethodInvocation* event), the policy selects a migration by copy. The constraint is necessary to assure that the destination Place is a gateway and not a simple node.

Let us show how this set of policies both at agent and middleware level can effectively rule at runtime the agent mobility behaviour. When the Configurer agent ends its code execution on a Place, the system automatically generates a *TaskCompletion* event.



```
inst oblig SysPol1
 on MethodInvocation(go,agents/invoker,args[]);
 subject  s = System;
 target t = agents/invoker;
 do
  t.gobyCopy.(args[0], closureClass, args[1]);
 when
  args[0].MiddlewareDATA.type =  "DefaultPlace";
```

Figure 4. An example of Middleware level policy.

The monitoring system delivers this notification to all the entities that have previously registered their interest in it, including the agent itself. The Configurer agent reacts to this notification by loading all the related policies from its profile. In this way, the `MobPol1` is also loaded and the agent tries to execute it. For this purpose, it first of all checks the constraint, testing if the size value of the *localPath* is zero. If the condition is not verified the agent neglects the policy, while if it is verified the agent enforces the policy and calls the *go()* method of the Adaptable Migration Interface. This call generates a *MethodInvocation* event and the adaptable migration component reacts to it by interpeting the set of policies defined for the Migration Service. As we have seen for the agent level, it first loads the `SysPol1` policy and then checks the constraint by verifying whether the destination Place is a gateway or not. If it is, the policy is enforced by selecting a migration by copy. If not, the policy is neglected and if there are no other active policies the default mechanism of migration by need is selected.

It is worth noting that a skilled user can invoke directly the right migration method provided by the Migration Service instead of the general method *go()* from the agent policy set.

The mobility behaviour so designed, both at agent and Middleware level, can be changed in any moment by simply modifying the set of policies without stopping agent or system execution. For example, to adapt the Configurer agent to an abnormal load of the remote place revealed by the agent during execution, the programmer can modify its migration strategy just by changing the `MobPol1` in its profile. The new policy (label a2 in Figure 3) changes the instant of migration (when) of the previous policy, and sets it at the end of the daily work of a machine, which is notified through an *EndWork* event. In this way the total time of exploration is enlarged, because the agent has to locally wait for this event in every place before migrating. However, the policy improves the local load and assures a consistent state of the remote Place during the configuration, by avoiding any kind of conflict between the agent updating task and the software execution task.

As a final remark, we can claim that the integration of the Ponder language in the SOMA framework offers programmers the twofold advantages of handling adaptive migration plans and adaptive migration mechanisms. In addition, the support of adaptable migration interfaces facilitates the transparent addition of new migration mechanisms in SOMA. Migration mechanisms can be easily plugged into the SOMA migration service and selected by simply defining appropriate middleware policies.

4. Conclusions

Component mobility requires appropriate software engineering techniques to support adaptive code relocation strategies and to manage the mobility behaviour at runtime with no impact on component implementation. A policy-based approach to mobility permits to express the mobility behaviour of components separately from the application logic at a different level of abstraction and provides the ability to dynamically change the migration strategy without changing the component's functionality. The integration of the Ponder language in the SOMA framework shows the feasibility of the approach.

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] D. Lange, M. Oshima, "Programming and Deploying Java Mobile Agents with Aglets", Addison Wesley, Menlo Park, CA, 1998.

[3] G. Goldszmidt, Y. Yemini, "Distributed Management by Delegation", *ICDCS'95 15th International Conference on Distributed Computing Systems*, IEEE Computer Society, Vancouver, British Columbia, 1995.

[4] A. Corradi, E. Lupu, R. Montanari, C. Stefanelli, "Policy Controlled Mobility", ICSE '01*Workshop on Software Engineering and Mobility*, IEEE Computer Society, Toronto, Canada, 2001.

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

[6] N. Damianou, et al., "The Ponder Policy Specification Language", *Proc. Policy Workshop 2001*, Springer Verlag, 2001.

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

[8] A. Corradi, N. Dulay, R. Montanari, C. Stefanelli, "Policy-driven Management of Mobile Agent Systems", *Proc. Second Policy Workshop*, Springer-Verlag, Bristol, Jan. '01.

[9] Meta Data Coalition. Open Information Model Version 1.0. http://www.mdcinfo.com/OIM/OIM10.html, August, 1999.