A Case Study in Role-based Agent Interactions to perform administrative tasks

GIACOMO CABRI, LUCA FERRARI, LETIZIA LEONARDI
Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia
Via Vignolese, 905 – 41100 Modena – ITALY
Phone: +39-059-2056190 – Fax: +39-059-2056126
E-mail: {cabri.giacomo, ferrari.luca, leonardi.letizia}@unimo.it

***Abstract* - Mobile agents are an emerging approach to develop distributed and parallel applications. They also can be used to develop enterprise applications, since they can act as active network components, and can execute on heterogeneous platforms and architectures. They also grant a high level of automatism, since they can be developed to perform one or more tasks without needing user interactions. In this paper we analyze the interaction issues of an agent based application for the automatic upgrade of Java programs, and propose a role-based approach to deal with the occurring interactions. Such an approach enables the development of flexible and reusable agent based applications, which can also be exploited by enterprise systems to simplify administrative tasks.**

***Keyword**: Agents, Roles, Interactions*

I. INTRODUCTION

A mobile agent is an autonomous software entity that performs its tasks without user interactions, and that can move to different environments during its execution. A mobile agent is an active network entity, where the term "active" indicates both the capabilities of adapting to the network environment and of executing in an autonomous way. This adaptability includes the capability of spontaneously moving to a particular node and run on it. This can be needed to found a particular resource and to use it in a cheaper way than the client-server paradigm. Thanks to their adaptable behavior and to their capability to run and move in a spontaneous way, mobile agents can be exploited to build distributed applications in network-based scenarios, such those related to enterprise systems. For instance exploiting mobility, an agent can move itself to an uncharged node, to perform load balancing. Furthermore, thanks to its portability, an agent can run on a very large set of platforms, without requiring users to change their legacy systems. Moreover, applications developed with mobile agents may be faster and more efficient than client/server ones [15], saving also bandwidth (important when using unstable or busy networks).

Mobile agents can be aggregate to build complex software systems (called MAS, Multi Agent System), in which every single agent is in charge of performing a task unit. After each agent has performed one of its tasks, it must communicate results to the other agents (or to a user), in a cooperative way. Moreover, during its life one agent is likely to interact with other agents or platforms in a competitive way, for instance to achieve resources. This scenario requires a new methodology to model agent interactions, both cooperative and competitive. The BRAIN (Behavioural Roles for Agent INteractions) framework [5] proposes an approach based on the concept of role to deal with interactions in agent-based application development.

A role is a set of capabilities, behavior and knowledge that an agent can assume and use to perform its tasks. Roles allow a separation between execution concerns, embedded into the role, and the mobility ones, embedded into the agent itself. The use of roles has several recognized advantages, particularly in the management of agent interactions, such as the already mentioned separation of concerns [7], reuse of solutions and experiences as a sort of design patterns [3], and locality in interactions [10]. Exploiting the role advantages, the BRAIN framework aims at covering the agent-based application development in different phases, and provide for (i) a model of interactions based on roles [8], (ii) an XML-based notation to describe the roles [6], and (iii) interaction infrastructures based on such model and notation, which enable agents to assume roles [10, 9].

In this paper we consider an application example to demonstrate how agents can be exploited in enterprise applications. In particular we consider a very common administrative task: the software update. Our case study considers a mobile agent based application used to automatically upgrade Java software. With this automatism the administrator must simply run the mobile agent and the latter will act as a scout on the network (either LAN or Internet) searching for an upgrade to download and install on its home node (i.e. the node from which the agent has been launched). During its life the "scout" agent interacts with a lot of services and systems, so the interactions must be carefully designed. We do this exploiting the BRAIN framework, thus using roles to define the interaction issues in the different cases.

The paper is organized as follow. Section 2 shows the motivations and the advantages of the use of roles, with a short introduction to role, role systems and the assumption/release process. Section 3 presents the case study, and its development following our role-based approach. Section 4 proposes some considerations about our approach. Section 5 briefly compares our approach with other ones. Finally, Section 6 reports the conclusions.

II. MOTIVATIONS

Roles can be thought as an abstraction layer between the agent itself and the environment: in fact, context-specific actions and knowledge are embedded into roles, untying agents (and their developers) from knowing or implementing them. The agents execute one of its tasks through a specific role, requiring a service like method invocation or variable content value. For example, as shown in Figure 1, an agent can use a role called "writer" to access to a database: the agent must simply ask its role for a service to obtain the database access. In this way both the agent and its developer are untied from knowing database particulars, like the DMBS type, commands, etc., because all these information are acquired with the role assumption

In this way, roles are similar to some design patterns for enterprise applications, such as the "Session Façade" for EJB [13], and like these patterns, roles increase the application scale up and reusability, simplifying the developing of complex applications based on mobile agents. Roles also allow security controls, since permissions needed to execute a task can be granted only to a specific role and not to the agent, and furthermore it is possible to develop security managers that allow/deny operations depending on the role the agent has assumed. For example, in Figure 1 a database security manager denies a write operation on the database if the role assumed is not the writer role.

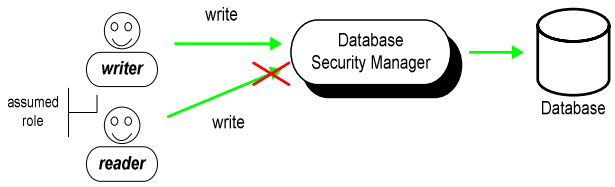

Figure 1 - An example of security check based on the role external visibility.

Other systems exploit the concept of role to implement access controls, in a similar way to the above one. In these ways roles are intended not only as a set of capabilities but also as a set of permissions (see the RBAC implementation [18]).

Thanks to roles, an agent can increase its adaptability to the execution environment. In fact, as specified above, one role is tied to the environment and not to agents, so one role can be used as an interaction interface between agents and the environments themselves (either other agents or platforms). This is an important issue, since enterprise systems could be very heterogeneous and agents should execute always and everywhere. Thanks to roles this is possible, since one role is in charge of supplying agents with knowledge and functionalities to interact with the different enterprise environments. Moreover, a role can be the implementation of local interaction policies between agents and the environment that hosts them, depending on the application context.

An agent can use roles thanks to a role system, an infrastructure that allows an agent to assume, use and release one or more roles. A role system must be dynamic, enabling the agent to assume and release roles at run-time. In addition, it must grant a high portability for roles so that they can be used by different agents at different time/situations. To increase the agent adaptability to the local environment, the role choosing process must be centered on semantic information, i.e. what the role does, and not how it does it. Several approaches are based on syntactic aspects, in other words on how a role does anything; this leads to an implicit static assumption/use/release process, that means that all the actions are coded into the agent source code. This implies that a change to the role the agent must assume, for example, requires a change in the agent source code. To avoid this problem we propose an approach based on semantic information, which is contained into an XML document, called role descriptor. Using XML documents grants high portability of the information, so that other agents, developed with other technologies, can use it too. Furthermore, XML documents can be written and managed easily by automatic tools such as other enterprise components, and this is another step toward the integration of the different components in enterprise systems.

Another important issue is the level of security granted by the use of roles during user's information transport. In fact, since a role is tied to the platform, and not to the agent, it implies that the agent cannot move with the role assumed. This means that, before a movement, the agent must release every role it has assumed, and, only then, it can move to another environment. After arriving at that environment it can then assume another role, even if equals to the one it has assumed on the previous environment, to continue its execution. But in this situation the role cannot be used to store temporary data, so every data that the agent has acquired using a role must be returned by the role to the agent before it is released. In this situation the agent cannot understand the data returned by the role, because it does not have the knowledge that is embedded into the role. Of course a malicious agent can perform introspection, to understand the data content, but that data could also be returned encrypted, so that only another role, developed to understand it, can understand and validate the content.

As a final note of this section we stress how the searching for/assuming process is strictly dependent of the role system itself. This means that an agent can obtain a different role subset of the available roles depending on security issues (e.g. the agent home host, the agent type and so on). For example one host can use a policy that allows an agent to find and assume roles depending on where the agent is coming from, or depending on the agent type (i.e. agent class type), and, at the same time, it can allow an agent to find all available roles if it is signed. Since these policies can be set in different ways on different hosts, their use grants a lot of flexibility to the host administrators, that can easily maintain security issue related to agents and services running on their host.

III. THE CASE STUDY

In this section we present the application considered as case study: an automatic Java-software upgrade system, developed using agents. What the application

must do is (i) starting a scout agent that analyzes the software version of the program to be updated either in an automatic way or by a user interaction; (ii) then the agent must search for a newer version of the software and, when found, (iii) takes the upgrade information back and install it on the user system. The latter step can be performed in two different ways, depending on the application policy. The first way uses the scout agent only to perform an upgrade search, leaving it to explore the Internet and storing location(s) of upgrades. After that it can return back to home and automatically download the best upgrade. The second way uses the scout agent to find and return home the upgrade, so the agent, when found the best upgrade, returns to home carrying it with itself. In the former case the agent can be exploited to search for the upgrade of one or more programs, so that after it has returned home, it can perform a multiple download. The second case, instead, should not be used to perform more than one software update acquiring, due to band-width saving. In fact, since in this case the agent must carry on with it the upgrade, the "weight" (i.e. object size) of the agent grows up every time the agent take a new upgrade, so the agent transfers will be slower and will use more bandwidth. For reason, in the following, we take into account particularly the first case (upgrade search and download from the user's host).

The searching process depends of the user's preferences. In fact the upgrade could be chosen by its version, release date, size, stability, and so on. Even the number of sites visited to find the upgrade depends on the user's preferences. For example a user can choose to get the first found upgrade, also if unstable, because it is strongly needed, or she can choose to get the smaller one (in term of size).

All the above application steps imply different interaction issues, because the agent must interact and negotiate with the local services (either system services or other agents), accordingly to the local policies and architectures. Developing an application such as the above requires that the agents' interactions must be taken carefully into account. In fact, during the application execution, the agent will run on different hosts and in different environments, interacting with them and with other agents. In such scenario the adaptability of the agent is the hardest problem that a developer must face. As shown in the next subsection, roles make this task easy, allowing a strong code reusability and modularization.

A. Automatic Java-software update system

In this section we explain how agents and roles can be exploited to make up an automatic Java-software upgrading system. As shown below, it is also possible to extend this application to non-Java software, thanks to the flexibility granted by the use of roles. The application details are explained in the following.

The first step is the launch of the scout agent, which is the agent that starts its execution on the user platform. This agent is in charge of finding the programs' state (i.e. version, latest build, etc.) and of searching for upgrades. At the beginning of its execution, the scout agent assumes its first role: JP_introspector (Java Programs introspector). This role is used to crop the other Java application state. This cropping can be done in two different ways: the former is an automatic way, the latter is a manual way based on user interactions. The automatic way implies that the agent, through the role, performs introspection on others Java software (maybe searching into the classpath), and collects their characteristics (version, build number, etc.). The manual way instead requires the user interaction to perform the data collect process. It depends on role implementation the way in which the agent collects programs data (see Figure 2, and Figure 3).

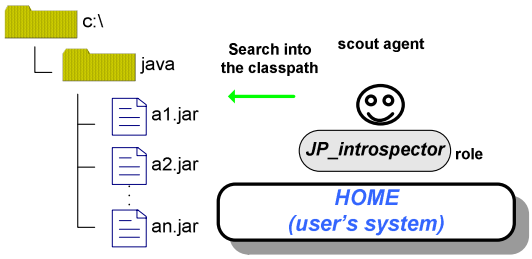

Figure 2 - A JP_introspector role implementation that performs data-collecting in an automatic way.

Note that different implementations of JP_introspector allow also determining which information is required by the application. For example, an implementation can collect both the version and build number; instead another implementation can collect simply the version number, and another one can collect both the version and the release date, and so on. The use of roles in these situations allows a light development of the agent, since it is the same in all situations. In other words, different behaviors are achieved by different role implementations, and without agent modifications. It is also possible to determine which information the role must collect using a local policy file, for example an XML file or a Java property file, so that the user, modifying it, can change autonomously the role behavior.

After the scout agent has collected the information required, it can move around the network to find an upgrade for the specified program. In this step interactions among agents and other agents/systems are more stressed that in the previous step. In fact, in the first step the agent executes always on the user system (simply called home in Figure 2, Figure 3). This simplifies the agent execution since it runs always in the same environment, thought as no-malicious. In the second step instead the agent must execute on different environments, perhaps malicious, so interactions must be taken more carefully into account.

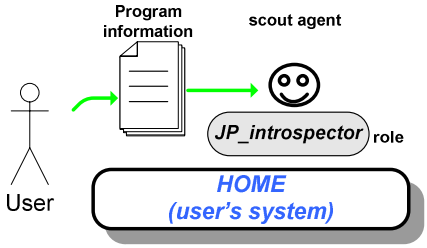

Figure 3 - A JP_introspector role implementation that requires the user interaction to collect programs information.

In this step the scout agent searches for and assumes a role that allows it to query a local software database where all upgrades are stored. Also this role, called upgrade_searcher, can have different implementations. These implementations depend on the local policies and also on the execution environments. For example, there can be two main different implementations, depending on the environment type. A first environment type can be whole modeled with agents. In such scenario the scout agent must interact with a local agent, called site agent, which is in charge of answering all the queries made by the *scout* agent through the through the *query* role. The situation is shown in Figure 4. In this scenario the scout agent passes the programs data (i.e. information on the searched software) to the site agent through its role. The site agent analyzes the information and performs a query to the software database. After this, the site agent returns the query result to the scout agent, which now knows if the upgrade is available on this node. If the upgrade is not available, the scout agent can discard the upgrade_searcher role and decide to move to another site. If the upgrade is available, the agent collects its information (see details below) and returns back to home.

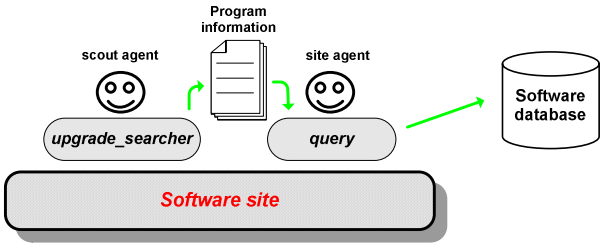

Figure 4 - The scout agent searches for the updates interacting with a site agent.

A second environment type can be modelled without other agents. In this scenario the scout agent interacts directly with the node system. Now the *upgrade_searcher* role is used to directly access the software database. Please note that in this case the *upgrade_searcher* role is different from the one used above. In fact, since here the agent must access directly the database (i.e. without interactions with the site agent), the *upgrade_searcher* must grant to the agent all capabilities needed to do it. This means that, in this situation, the *upgrade_searcher* role has the capabilities of the previous query role, used to access the database. Please note that, even if the two *upgrade_searcher* role implementations showed in Figure 4 and Figure 5 are different, in both cases the name of the role is the same. This is very important and stresses how roles grant a separation of concerns: the scout agent always searches for, and assumes, a role with the *upgrade_searcher* property, without knowing which implementation of the role will be used exactly (i.e. the one who interacts with the site agent, or the one which access directly to the database). The fact the *upgrade_searcher* role either access directly to the database or interacts with another agent is totally transparent to the scout agent agent itself, and also to its developer. In both cases the agent does not need to know details to the access procedure (for example the JDBC driver, the SQL statements, etc.). Thanks to the flexibility introduced by roles, also enterprise and legacy systems can be exploited by agents. It is not important how old is the database system, or how the software is stored, since roles provide agents all required methods to access it (Figure 5).

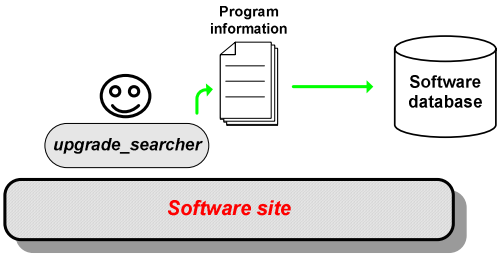

Figure 5 - The scout agent searches for the updates interacting directly with the software database.

Also in this situation the agent can move itself to another site if the required upgrade is not available.

The above considerations emphasize the code reusability granted by roles: as for the previous step, the agent does not require any modification, since its adaptability to different execution environments relies on the role implementation. It does not matter with which entity the scout agent must interact, since it must simply use the *upgrade_searcher* role to perform its tasks.

After the scout agent has found the upgrade it must collect it. Accordingly to the previous section, the agent does not acquire the software in this step, but simply keep a trace of the availability of an upgrade at this site.

The format of the upgrade depends also on the base software available at the home node. For example, if the user has the source codes the upgrade can be simply a patch; moreover the user can specify what format the scout agent must search for (jar, zip, etc.).

Once the scout agent has found the upgrade it can return home, download and apply it. This can be done simply substituting the original jar or the sources and invoking a recompilation (for example using Ant [1]). To perform this step the agent assumes another role: updater (Figure 6). This role is needed to separate the agent from the update specific actions, depending also on user preferences and system issues (e.g. file trees). When the scout agent returns home, it assumes the upgrader role, and starts the upgrade action. The upgrader role is in charge also of performing security checks. A lot of update tools, such as those provided with today's operating systems, uses a list of trusted sites to perform downloads, but what about agents? A good solution is the use of a trusted site list, a list which contains a set of trusted sites, which the agent enumerates before its search trip. On this enumeration the agent can build a mobility logic, which means a route among hosts. The trusted site list can be built either manually from the user, depending of her preferences, or compounding other lists in a similar way of the Web of trust used in PGP/GPG key rings. The use of trusted site list is useful not only to build the mobility-logic, but also to grant the agent the capability to understand if an upgrade can be performed in that particular moment. For example imagine that during the agent trip on the network, the user decide to exclude a specific site from her trusted site list. Since the agent has

no capabilities of knowing this exclusion (because it is not connected to the home node) it could decide to download software from a not yet trusted site, and this cannot happen. To avoid these situations the upgrader role must read the trusted site list each time before starting an upgrade.

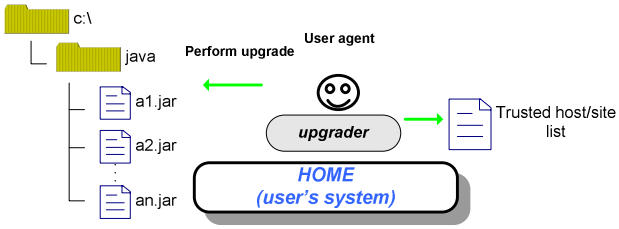

Figure 6 - The scout agent comes back and assumes the upgrader role to perform upgrades.

To better explain how roles are spread on nodes, Figure 7 shows the deployment diagram of our application. In that figure is shown how roles are tied to the local context: JP_introspector and upgrader belong to the home environment, while upgrader_searcher and query role belong to each software site. As already explained, since each role is tied to a network node, it is also tied to the node local policy. So every user (or administrator) can set up a specific policy on her HOME node to perform particular operations, like the check with the trusted host list mentioned above.

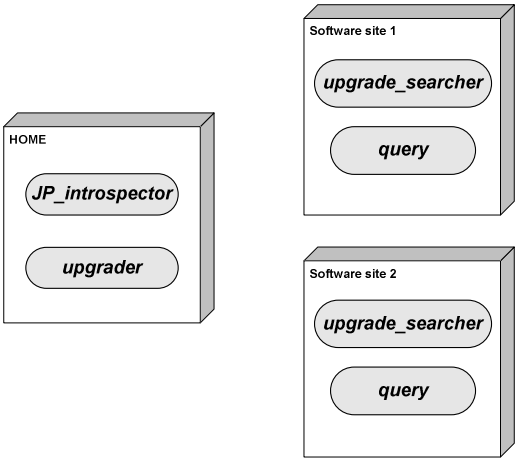

Figure 7 - Distribution of roles among network nodes.

B. *Discussion*

Now that the whole application has been explained, let us make some considerations about what the use of roles grants. First, note that the scout agent is in charge only of visiting the software sites, which means it must only concern about movements. In other words, the movement logic is embedded into the agent, while the interaction logic into the roles. Starting from this consideration we can point out that roles can be developed in a separated way from agents. The role developer does not know anything about how the next host to visit is chosen by the agent, while the agent developer disregards about how interactions are managed by roles: this grants separation of concerns. For example, the agent could choose the next hop by a WSDL approach or by using a static precompiled host list, but this is not important for the role developer.

Second, note that, since the application logic is wrapped by roles, it is very simple and fast to change the interaction logic, simply by changing the roles that the agent assumes. This also allows reusing the movement logic, maybe because it is tested and works well. For example the application can simply turn into a document upgrader: instead of upgrading software it can search for new versions of important document (such as ISO standard release).

An important aspect to emphasize is continue information-passing through different roles assumed by the same agent. In the first step, for example, the *JP_introspector* collects data that is then used by *upgrade_searcher*. Something similar happens in the last step, where the upgrader role uses data collected by the upgrade searcher. Since roles are tied to the local context, that means that an agent cannot move itself while it has a role, how can roles exchange data? A common data storage space must be provided by the agent itself. The simplest solution is to use a hash map to store information. It is important to note that the roles are developed into application contexts, so roles are related the one to the other. This means that the JP_introspector role knows that the agent will probably assume the upgrade_searcher as the next role, so it can format information for that role.

The idea of the hash map is simple: every execution context (i.e. every role set developed for the same application) stores all the data into an entry with a specified key. For example, all the roles used in the above application can store all data into an entry with the key SoftwareUpgrade. Every role will search into the hash specifying that key and will find the data. Furthermore, every entry can be another hash map to perform a better data organization. The hash map is useful when the agent must assume more roles belonging to different contexts. In this way every role has a separate store space and will not clash with the other roles.

Note that, even if there are no roles that provide the data storage space, all the data can be maintained safe from agent introspection or from a malicious role. In fact, since the roles are developed in the same context, they must use a particular store format, such as an encrypted one. In this way all roles belonging to the same context (and which know the data format) can interpret the data, and at the same time every entity that does not belong to the application context (included the agent) cannot.

It should be clear now how useful roles are for the scout agent, but what about the site agent in Figure 4? As shown in that figure, that agent uses a role called query role to interact with the software data and database, but it seems a non-sense since that agent is "static" (i.e., it does not move among different host). So, why also that agent must use roles, instead of use an embedded behavior? Of course the use of roles grants a better scale up and an easy adaptability to application changes, but the main reason is about the coding format of data between agents. As already described in the previous paragraph, roles involved in the same application context must be designed (and developed) together, so the query role can dialogue with the upgrade_searcher role and vice versa. A

change to one role must influence the other role, but not the agents that use them. On the contrary, if the agent uses an embedded behavior, a change to the upgrade searching logic implies a change on the site agent and maybe a complete re-design of it.

The above considerations lead to another conclusion: roles can be administrated in a separated way from agents. Of course if an agent cannot find the role it is searching for, it cannot perform its tasks, and it is for this reason that the administration of roles is not completely independent of the agent's one. But the agent platform administrator can manage roles and agents in a separate way, for example updating the agent to a new version and maintaining the same role or vice versa. This enforces the separation of concerns and the capability of specializing the agent for movements and the role for the application scenario.

Our application is different from the Update Tool for Java [2] developed from IBM Alphaworks, since it is really automatic and does not need source code modification to perform the upgrade. In fact the Update Tool for Java requires a manual user interaction to perform the upgrades and even if it is not mandatory, an application code restyling to make easier the update process. Furthermore the above tool requires the user to search for and download the update, while our system does it autonomously. Nevertheless our system requires a Mobile Agent Platform executing at the home host and at download hosts, so that the agent can move among those hosts. This is a little advantage because the same platform used by this application can be used for different applications and/or services, as a mail service [12], so Mobile Agent Platforms, instead of other services, do not belong to a single service type.

IV. Comparison with other Approaches

In this section we sketch the comparison of our approach with other ones in the management of agent interactions. We cannot report many details because of the page limitation, but we will try to provide the reader with the essential issues.

The first considered approach is the traditional OO-based, which implies the definition of methods in the agent code, in order to interact with other agents (or entities). The mostly adopted model is the message-passing one, usually exploited in a "procedure call" fashion, i.e., the agent invokes a method and expects an answer as return value. Even if this approach is the simplest one, its main limitation is that the specific features concerning the role played by the agent are not separated from the general features, for instance from the mobility or the planning features. This leads to some important drawbacks:

Interactions are more object-oriented than agent-oriented, losing the abstraction degree proper of the agent paradigm.

The software is more difficult to be maintained, because modifications or additions often influence the (i.e. both role and agent code since the latter must use the former), and cannot be isolated in the related sections.

From the Internet site point of view, specific site-dependent code of actions cannot be provided, and agents must embody the code from the beginning. Moreover, it is not possible to enforce local laws and to control the interactions among them.

The second considered approach is based on the Aspect Oriented Programming (AOP), which, even if it not designed in connection with roles, seems to provide interesting mechanisms to support the management of roles for agents [11, 16]. AOP starts from the consideration that there are behaviors and functionalities that are orthogonal to the algorithmic parts of the objects [17]. So, it proposes the separate definition of components and aspects, to be joined together by an appropriate compiler (the Aspect Weaver), which produces the final program. The separation of concerns introduced by AOP permits to distinguish the algorithmic issues from the behavioral issues. Since an aspect is a property that cannot be encapsulated in a stand-alone entity, but rather affects the behavior of components, it is evident the similarity with a role. Even if the AOP approach is similar to ours, in our opinion it has some limitations:

First, the role/aspect must know the class that is going to modify, strictly coupling agents (classes) and role (aspects).

As a consequence of the first point, this approach lacks flexibility in the definition and usage of aspects, and this is due to the fact that AOP focuses on software development rather than addressing the issues of dynamic and wide-open environments, such as the ones considered in the BRAIN project.

Finally, interoperability among agents of different applications is hard to be achieved, since this approach does not provide an adequate uncoupling of roles from agents.

Conclusions

This paper has presented a case study application based on mobile agents, whose interaction issues have been faced by means of a role-based approach. We have shown the advantages of the use of agents for automatic tasks, such as the administrative ones. Thanks to roles this application can be developed in a faster way and with a high degree of scalability. It is for these reasons that roles are probably the better solution for a developing and execution environment for such applications. Furthermore the most important thing is that roles provide agents with a better adaptability, so that they can suit the policies and mechanisms of the local environment. This makes mobile agents a good system to exploit dynamic services, like e-commerce or enterprise applications too.

As a final note we stress how roles propose a new paradigm, also exploited into other computer-science subjects [4], that extends the agent paradigm introducing a new way to build scalable applications. As already mentioned in this paper, since roles embed the interaction-logic, agents can be developed caring only about movements, and roles become important such as the agent themselves. This leads to a fast developing time and on more specific and separated developing phases.

ACKNOWLEDGMENTS

Work supported by the NOKIA Research Center of Boston, by the Italian Research Council (CNR) within the project "Progetto Strategico IS-MANET, Infrastructures for Mobile ad-hoc Networks", and also by Italian MIUR.

REFERENCES

[1] The Apache Ant Project: http://ant.apache.org/

[2] Update Tool for Java, IBM Alphaworks, http://alphaworks.ibm.com/tech/updatetool4j

[3] Y. Aridor, D. Lange, "Agent Design Pattern: Elements of Agent Application design", in Proceedings of the International Conference on Autonomous Agents, ACM Press, 1998.

[4] Dirk Baumer, Dirk Riehle,Wolf Siberski, Martina Wulf, "The Role Object Patterm", Pattern Languages of Programming conference, 1997, Monticello, Illinois, USA

[5] The BRAIN project, http://polaris.ing.unimo.it/MOON/BRAIN/index.html

[6] G. Cabri, L. Leonardi, F. Zambonelli, "XRole: XML Roles for Agent Interaction", in Proceedings of the 3rd International Symposium "From Agent Theory to Agent Implementation", at the 16th European Meeting on Cybernetics and Systems Research (EMCSR 2002), Wien, April 2002.

[7] G. Cabri, L. Leonardi, F. Zambonelli, "Separation of Concerns in Agent Applications by Roles", in Proceedings of the 2nd International Workshop on Aspect Oriented Programming for Distributed Computing Systems (AOPDCS 2002), at the International Conference on Distributed Computing Systems (ICDCS 2002), Wien, July 2002

[8] G. Cabri, L. Leonardi, F. Zambonelli, "Modeling Role-based Interactions for Agents", The Workshop on Agent-oriented methodologies, at the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), Seattle, Washington, USA, November 2002

[9] G. Cabri, L. Ferrari, L. Leonardi, "Enabling Mobile Agents to Dynamically Assume Roles", The 2003 ACM International Symposium on Applied Computing (SAC), Melbourne, Florida, USA, March 2003.

[10] G. Cabri, L. Leonardi, F. Zambonelli, "Implementing Role-based Interactions for Internet Agents", 2003 International Symposium on Applications and the Internet, Orlando (USA), January 2003.

[11] Communication of the ACM, Special Issue on Aspect Oriented Programming, Vol. 33, No. 10, October 2001.

[12] S. Chiba, "Flyngware: an Email Based Mobile Agent System", position paper available at http://www.csg.is.titech.ac.jp/~muga/flyingware/flyingware.html

[13] Session Façade Design Patter : http://search.java.sun.com/search/java/?qt=facade

[14] Sun Microsystem Enterprise Java Beans : http://java.sun.com/ejb

[15] Raul Jha, Shrydar Ier, "Performance Evaluation of Mobile Agents for E-Commerce Applications", International Conference on High Performance Computing, Hyderabad, India, Dec 2001.

[16] E. A. Kendall, "Role Modelling for Agent Systems Analysis, Design and Implementation", IEEE Concurrency, Vol. 8, No. 2, pp. 34-41, April-June 2000.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin, "Aspect-Oriented Programming", in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Finland, June 1997.

[18] R. S. Sandhu, E. J. Coyne, H. L. FeinStein, C. E. Youman, "Role-based Access Control Models", IEEE Computer, Vol. 20, No. 2, pp. 38-47, 1996.

[19] N. Ubayashi, T. Tamai, "RoleEP: role based evolutionary programming for cooperative mobile agent applications", Proceedings of the International Symposium on Principles of Software Evolution, 2000.