Dynamic Deployment of Services on Mobile Agents Systems

Nuno Santos, Paulo Marques and Luis Silva

CISUC, University of Coimbra, Portugal
{nsantos, pmarques, luis}@dei.uc.pt

Abstract. Mobile agents are a very interesting paradigm for structuring distributed applications. By using them, distributed applications can implement part of their functionality as a set of loosely-coupled components, which are able to migrate. This allows the creation of very modular and extensible designs since it is quite easy to change the functionality that is available in each node of a large distributed system. Nevertheless, most mobile agent platforms are not extensible themselves, being implemented as monolithic entities. This has serious implications since the platform does not accommodate changes after being deployed, making it hard to introduce new functionality, correct implementation errors or introduce performance enhancements without stopping and recompiling the whole infrastructure. In this paper we describe a component-based framework for mobile agents with support for dynamic reconfiguration, so that components can be added, removed and reconfigured at run time, with minimal disruption to the application.

Keywords: Dynamic Reconfiguration, Mobile Agents, Dynamic Deployment, Services, Java

1 Introduction

In recent years there has been a growing interest in component-based [1] middleware systems, mainly due to their greater flexibility over monolithic systems. Components simplify application development and maintenance, allowing them to be adapted more quickly to changing conditions.

Mobile agents are a software development paradigm with many of the same advantages of components. They are well-defined entities, with a clear boundary between internal state and external interfaces, which allows them to be used as reusable units of software. They have the added advantage of being able to migrate between hosts of a distributed application. As some authors point out [2], this ability can be used to achieve greater flexibility over traditional middleware. A distributed application based on mobile agents can be reconfigured dynamically with minimal disruption to the whole application, by terminating some of the agents and dispatching new ones to the nodes of the application requiring an update.

But in most middleware for mobile agents systems this flexibility ends at the level of the platform. Usually, it is implemented in a monolithic way, with little or no provision for reconfiguration. This has several disadvantages. On one hand, the platform is hard to deploy, because a monolithic architecture does not integrate well with other applications.

On the other hand, it has poor or no extensibility. For instance, adding a new inter-agent communication service to the system normally forces the whole platform to be recompiled and redeployed. One common workaround is to use static agents to extend the platform, providing system level services. But this is an abuse of the mobile agent concept, which is not practical to implementing system level functionality, resulting in cumbersome and inefficient designs.

In previous work [3] we have identified this lack of extensibility at the platform level as an important limitation to a greater use of the mobile agent paradigm. To address this limitation, a component-based middleware for mobile agents (the M&M framework) was implemented using Java [4, 5]. It is structured as a set of JavaBeans [6] components, with a base `Mobility` component supporting a core set of functionality, like mobility, security and extensibility. This component can easily be added to any application, just like any other JavaBeans component, making it very simple to create mobile agent enabled applications.

The functionality of the `Mobility` component can be extended by plugging new services into its extensibility layer (Figure 1). *Services* are also JavaBeans components. After being plugged, services became available to any client that can interact with the extensibility layer, including the application, the mobile agents and other services.

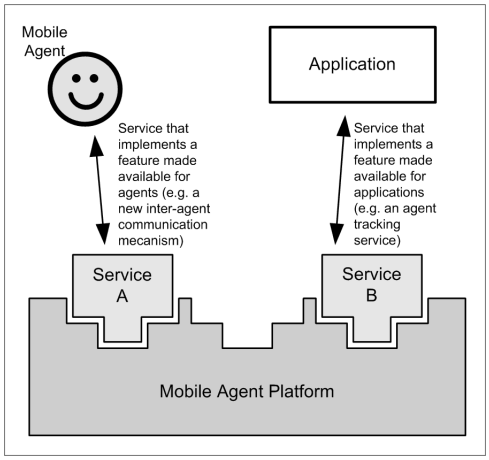

Fig. 1. Services can extend the M&M middleware, by providing new functionality for mobile agents and to the application.

Although this model has been used with success to implement and deploy several services, like agent tracking, inter-agent communication and disconnected computing [7], it is still limited as it only supports static reconfiguration. Adding, removing and upgrading services forces the application to be stopped and restarted, which represents a downtime

that is undesirable in some situations. This led us to improve the model with support for dynamic reconfiguration.

In this paper it is described how dynamic reconfiguration of services is implemented on the M&M framework. Services can be added, removed and upgraded at run time, with little or no disruption to the application or to the mobile agents that are running and using those services. It is also described how services can be remotely reconfigured by using RMI [8] or mobile agents.

The remainder of this paper is organized as follows. Section 2 describes the implementation of dynamic services in the middleware. Section 3 explains several aspects of service management, like lifecycle, configuration (both local and remote) and security. Section 4 provides an overview of related work. Section 5 describes some limitations of this work and provides ideas for future work. Finally, Section 6 concludes the paper.

2 Supporting Dynamic Services

An architecture supporting dynamic reconfiguration must satisfy some requirements to ensure that reconfigurations can be properly specified and carried out. In turn, this makes it desirable to build the middleware on top of a platform with support for dynamic linking and reflection. Fortunately, Java has good support for both features, which made it possible to implement dynamic services using only the standard interfaces available on the Java platform.

2.1 Requirements for Dynamic Reconfiguration

As identified in [9], there are three basic issues that must be addressed by any system that supports dynamic reconfiguration:

- **Specification and management of change** There must be a way to describe the changes that are to be made and the conditions required by them. This includes specifying software and hardware requirements for a component, and the external relations and bindings with other components.
- **Preservation of Consistency** After reconfiguration, the system should be working properly. The components should be in a mutually consistent state and no bindings should be broken.
- **Minimum disruption** The objective of dynamic reconfiguration is to avoid disruption of the service provided by the application. In particular, it is important that parts of the application that are not being reconfigured are not disrupted.

This work focuses only the last two points, that is, supporting upgrades with minimal disruption to the system, while preserving its consistency. The requirements and bindings between the services that have been developed for the M&M middleware are usually

simple enough so that they either can be encoded directly on the service code or can be done manually. Therefore, it was not important to provide support for that at this moment. In future work that requirement will be addressed.

2.2 Java Support for Dynamic Reconfiguration

The Java platform has some very interesting features for implementing dynamic reconfiguration, like dynamic class loading [10] and reflection [5].

Class loaders are the entities responsible for loading and linking classes in the Java Virtual Machine [11], providing separate name spaces for the classes they load. They are also the unit of code loading and unloading. When the class loader and the classes defined by it are no longer referenced by external classes, the garbage collector is free to unload them. Class loaders are first-class entities and can be created by programmers, allowing them some control over the process of loading new classes into the virtual machine. They are used by browsers to load applets from the network, and by application servers and servlet engines to load new components. In this work, class loaders are used to load, unload and reload services in the M&M middleware.

Reflection is a feature of the Java platform that makes the internal structure of a class or interface available to the programmer. This meta-information includes, among other things, the names of the methods, of the fields, interfaces implemented and super-classes. This is important for supporting dynamic reconfiguration, allowing a framework to interact at run time with components that are unknown at compile time.

2.3 Anatomy of a Service

From a programming point of view, a service can be viewed as a JavaBeans component, which must contain the following:

A `ServiceDescriptor` An object used to describe a service, defining its identity. It contains the name, version and the service interface name. Two services are equal if their service descriptors are equal. Two services are compatible if they have the same name and service interface. This allows services to be upgraded, as it is explained below.

An implementation of the `ServiceProvider` abstract class This is used by the middleware to interact with the service. It provides methods to create new instances, change the configuration and obtain the service descriptor of a service.

A service interface and its implementation The service interface is derived from the well-known `Service` interface. It is how the clients interact with a service, and defines the functionality specific to the service. This means that each service will have a specific interface that is known by the clients but not by the middleware. The service must also provide an implementation of this interface.

These requirements play no role in the support for dynamic reconfiguration, and are needed just to create well-known interfaces between the service and the exterior, i.e. the middleware and the service clients.

2.4 Implementation

To make it possible to load and unload services at run time, they cannot be loaded in the same class loader as the application (Figure 2). It is necessary to use a separate class loader for each service (Figure 3). This isolates services from each other, making them independent units. It is even possible to execute different versions of a service side-by-side. This might be necessary when different versions of the service are not backwards compatible.

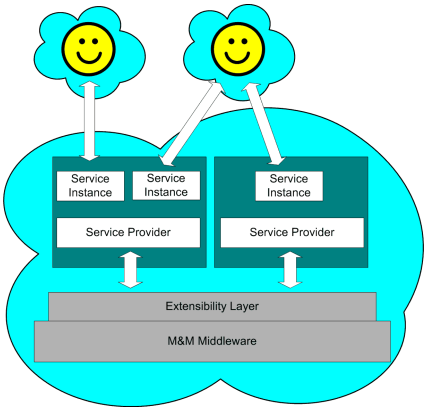

Fig. 2. Static Services. The services are loaded by the same `ClassLoader` as the rest of the middleware. Although the service can be stopped, it is not possible to unload its classes without unloading the whole middleware.

But using separate class loaders is not enough for enabling services to be unloaded and replaced. For a service and all its classes to be unloaded, its class loader must be unloaded. A user-defined class loader is represented by an object in the Java Virtual Machine and is subjected to the same rules as any other object. This means that only the garbage collector can clean it and this happens only when there are no more strong references to it [11]. This means that there can be no strong references to classes loaded by the service, because they all have a direct reference to the class loader. Also, there can be no strong references to objects of those classes, because they contain direct references to its class type. Therefore, the service must be loaded and handled in a way as to ensure that it is possible to destroy all strong references to its class loader, to its classes and to its classes's instances that may exist from the middleware, from the clients or from the application.

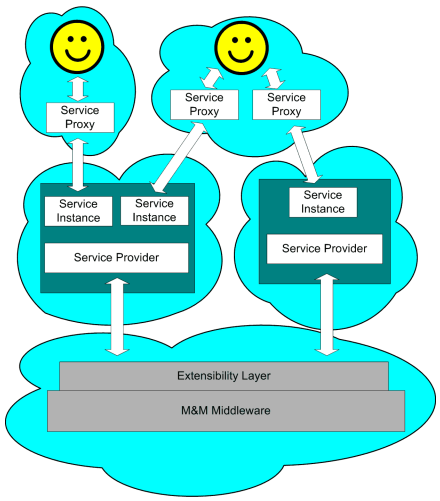

Fig. 3. Dynamic Services. Each service is loaded by its own `ClassLoader`. The `ServiceManager` module can load and unload services.

Dealing with the direct links to an object is straightforward. It is only necessary to keep track of all references that exist to it (it is explained below how this is possible by using proxies) and put them to null when it is time to unload.

Dealing with direct links to classes is not as simple. To see why, it is necessary to understand how objects of type `Class` reference each other. When loaded, a class contains only symbolic references to the other classes it uses [12]. Later, sometime before the first access from that class to the referenced classes, the system resolves the symbolic reference into a direct reference. A direct reference is a pointer to an actual class object defined in the JVM and cannot be changed.

For our purposes, this is a problem if classes outside the service (middleware or clients) obtain direct references to the service classes. Once this happens, only when the outside class that has the direct reference is unloaded, will it be possible to unload the service. Most of the times, this means shutting down the application. Therefore, direct references from external classes to service classes must be avoided.

There is the possibility of establishing direct references in the two interfaces services export: to the middleware and to the clients. Direct references in the first interface are avoided by using polymorphic invocation, and in the second by using proxies.

Interaction between the middleware and the service is done in a similar way as happens with applets and servlets. In these situations, there is a well-known abstract class from which the main class of applets and servlets must inherit. This class is then used for all interactions. Services must also derive a class from the well-known `ServiceProvider` abstract class. When loading the service, the middleware instanti-

ates this class and uses it for all interactions with the service, but always using a reference of the type of the `ServiceProvider` abstract class. Polymorphism takes care of forwarding the method invocations made using the `ServiceProvider` reference to the service class. In this way, a direct reference is never created from the middleware classes to the service main class.

A similar method cannot be used for service clients. The interface between clients and the service is known by the clients that use the service, otherwise they could not have been programmed to use it. But the middleware does not know it, or else this would not be dynamic deployment. Therefore, this interface class definition cannot be expected to be in the middleware codebase. But the client must use a reference to the service-specific interface, so the class must be accessible to the client somehow. The service always carries this class definition, as it is part of the service classes. Being in the service codebase, this class will be defined in the service class loader. If the client were to use this particular class, there would be a direct reference between client and service classes. Another problem would occur if the service were not installed in the middleware when a client that references it started to execute. In this situation, the client should be able to execute normally, until it requires the service. Then, it should be informed that the service is not present, being allowed to adapt to the situation somehow. But if the service class interface is not present, when the references made by the client classes to the service interface are resolved, a `ClassNotFoundException` exception is thrown and the execution of the client is stopped abruptly, giving it no chance to adapt.

The solution to both problems is to require the client to carry its own definition of the service interface class, so that this class is defined by the client class loader. This means that there will be two incompatible definitions of the service interface class, one defined by the client's class loader, and another defined by the service's class loader. This creates another problem. The service, when requested to create a service instance, creates an object that implements the interface defined by the service's class loader. As this interface is not type-compatible with the one defined by the client's class loader, a direct reference cannot be given to the client. This problem is solved by using the **Dynamic Proxy API** [13]. The middleware gives the client a dynamic proxy, which implements the interface class defined in the client. Internally, the invocation handler for the proxy uses reflection to invoke the service object. This avoids type-incompatibilities and direct references.

We will explain in more detail how loading, unloading and upgrading of services is performed.

Loading a new service For loading a new service, the `ServiceManager` needs to know where the Jar file with the service is, and what is the name of the service class that implements `ServiceProvider`. It will create a new class loader for this service, using it to load and instantiate the service provider.

Creating a new instance When a client requests a new instance of a service, the `ServiceManager` delegates this request to the `ServiceProvider`. Internally, the service is free to create a new instance for each client or to re-use the same instance for several clients. When the `ServiceManager` obtains the service instance, it creates a dynamic proxy using the service interface class defined in the codebase of the client (so that it is type-compatible with the client references), and wraps the service interface inside the proxy. It is this proxy that is returned to the client.

The `ServiceManager` keeps weak references [11] to all exported proxies. This is necessary to unload and upgrade services. Weak references are used so that the garbage collector can reclaim service instances that are no longer in use by clients. If strong references were used, the `ServiceManager` would keep those instances alive, even after the clients having stopped using them.

Unloading a service To unload a service it is necessary to release all references to instances of the service's classes, objects of the type `Class` representing classes from the service, and finally the service class loader. When this happens, the garbage collector is free to release the class loader and any classes defined by it.

The only places where these references might exist are in the `ServiceManager` (a reference to the class loader) and in the proxies that were exported to the clients (references to service class objects). To unload a service, the `ServiceManager` invalidates all exported proxies, making them release the reference to the service object. As a consequence, the next time the client tries to use the service, it will fail and receive an exception informing that the service is no longer available.

After all proxies to the service are invalidated, the `ServiceManager` releases all internal references to the service class loader and to the service classes. After this, there are no more references from the client or from the middleware to the service and the garbage collector is free to unload the service.

Upgrading It would be possible to upgrade a service by first unloading the old version and then loading a new one, in two separate operations. But this is not transparent to clients, which would abruptly loose access to the old service instance and to all the associated state. The clients would fail or, if they were ready to deal with the situation, would have to request a new instance of the service, loosing all the state associated with the previous one.

To minimize the disruption to the system it is important that the upgrade be transparent to service clients. This is a hard problem, as pointed in [10]. Upgrading a class at run time requires the following to be done:

– Migrate the state of the objects of the old class into objects of the new class. If the schema of the classes is different, it is necessary to adapt the state to conform to the new schema.

– Mapping the static fields of the old class into the new class.
– Ensure that no client is executing a method from the class that is being upgraded.

The first and second problems are hard to solve without intimate knowledge of the classes being upgraded. For instance, it is not clear how the middleware should perform the upgrade if the schema of the classes is not the same, or if the service has active threads or acquired locks. This must be the responsibility of the service developer, who knows the classes and the service behavior. Therefore, it was decided that for a service to be upgradable it must implement methods to save and restore its state and the state of all running instances into a canonical representation, as suggested in [14].

The third problem is dealt by using a read-write lock for each service. All proxies of a service must obtain read access before executing any method of the service. When an upgrade is to be performed, the `ServiceManager` acquires a write lock, so that all service clients are kept outside from service code. This works because the only entry point that clients have into the service is the exported proxy, so we can be sure that no client is executing code when the `ServiceManager` has write access.

The procedure for upgrading a service is as follows:

– The new version of the service is loaded by a new class loader.
– The `ServiceManager` acquires a write lock, preventing clients from accessing the service methods.
– The old version is given a chance to shutdown and save all the state of the service (global and instance) into a canonical representation.
– The new service is given the global state of the old version, so that it may place itself in a state consistent with the old version.
– For each proxy exported by the old service, a new instance of the service is created using the state of the corresponding old instance. The reference to the old implementation is replaced by the reference to the new implementation.
– The write lock is released, allowing clients to continue to work with the service.

The service is not forced to use serialization to save its state. This would not allow passing live objects, like threads, and would impose an unnecessary performance penalty. For objects of classes that are defined in the middleware codebase, it is enough to pass the reference to the object.

Nevertheless, serialization may still be useful for objects of classes defined in the service codebase whose schema remains the same during the upgrade. In this situation, although the schema of the class contained in the new service is the same as the one of the class of the old service, these classes are type incompatible because they were loaded by different class loaders. Therefore, objects of this class cannot be passed by reference. But it is possible to pass them by using serialization.

It is up to the service developer to choose how the state should be passed during the upgrade. For instance, it is possible to pass part of the state using references and part using serialization.

2.5 Restrictions on Services and Their Clients

In the previous discussion some of the conditions that services must satisfy were briefly mentioned. In short, they are:

- All interaction between the client and the service must pass through the service interface.
- The arguments or return values used by the methods defined in the service interface cannot be of classes defined in the service codebase.
- Service clients must have the service interface class definition in their codebase.

3 Service Management

The previous section described how services are supported and managed internally by the middleware. This section describes how services interact with external entities, like the application, the clients and other software components. It also describes how administrative tasks, like installing, configuring and using services, are performed and how security is implemented.

3.1 Service Lifecycle

Services have an associated lifecycle with the following states:

Absent The service code is not present on the application.

Deployed The service code is installed on the application, but the service is not available to clients.

Available The service is installed and available to clients.

There are two ways to obtain information about the state of services at run time: direct method invocation on the `ServiceManager`, and listening for service lifecycle events issued by the `ServiceManager`, which are issued when the state of a service changes. Any component or client with access to the `ServiceManager` and with proper authorization can listen for these events. This includes the application, other services and mobile agents.

3.2 Security

The M&M framework supports authentication and authorization of users [15]. Therefore, protecting the `ServiceManager` from the users of the framework was just a matter of defining a new java permission [16]. Using this permission, the administrator can specify what type of operations a user can do with a certain service. There are four different types of actions: obtaining information about services, requesting and using them, reconfiguring the extensibility layer, reconfiguring the services, and listing for service lifecycle events.

Another security issue is deciding what permissions should be granted to the deployed services. If the services were installed at design time, the administrator could check if the code is trusted or not. Trusted services would be deployed with full permissions while untrusted services would not be deployed. But when services are installed at run time, there is no way for an administrator to check if the service is trusted. There is a similar problem with applets and mobile agents.

In the M&M framework all threads run under the identity of a user. When a client calls a service, there are two sets of permissions involved: the permissions from the caller and the permissions from the user that deployed the service. It is not obvious what permissions should be used to execute the invocation. Some possibilities are to execute it with the permissions of the caller, with the permissions of the service deployer or with a mixture of both.

If an invocation is executed with the permissions of the caller, the service deployer can take advantage of this situation to execute code of its own under a less restricted security policy. Suppose that the service deployer does not have a permission A. It can install a service with code that requires that permission, and wait for a client with permission A to call the service before executing that code. This situation cannot be allowed.

The solution is to execute the method invocation using the interception of the permissions of the service deployer and of the client, and to allow the service to use a privileged block, when necessary, in order to regain the full set of its permissions. In this way, there is no danger of clients with less privileges calling the service. It is true that, while executing the service, the thread of the client might have its privileges elevated if the service uses a privileged block. But this is not a security breach, since the client cannot execute code of its own with the elevated privileges.

3.3 Service Configuration

Most services will need to have a configuration that can be changed at run time. If the service were well-known by the middleware at design time, it would be possible to define standard JavaBeans properties on the Service Provider and use them to change the configuration. But with dynamic services the Service Provider is not known at compile time by the middleware. Therefore, it is necessary to use more indirect ways of configuring a service. Two mechanisms are supported:

Reflection Using Java support for reflection, it is possible to discover the properties defined on a certain class. This way, the `ServiceManager` can still discover which properties are defined on the Service Provider, and make them available to the application. This allows the application to get or set the value of any property of a service at run time.

Messages Another way to configure a service is by sending messages to it. This is more general than properties, because the Service Provider is free to define the messages it wants and to react to them in any way. For instance, a service can listen for a message that tells it to shutdown. `ServiceManager` supports both targeted messages, which are received only by the target service, and broadcasts, which are received by all services.

3.4 Configuration Interfaces

`ServiceManager` is the component responsible for all operations related to services. It is a local interface, so only the application has access to it. But it is also important to provide other ways to configure the services, so that it is possible to manage the application remotely. Two external interfaces have been implemented. These interfaces act as bridges to the `ServiceManager`: one for RMI clients, implemented as a standard RMI server object exported by the middleware, and one for mobile agents, implemented as a service.

These two interfaces fully support the operations of the `ServiceManager`. The only difference is that when deploying new services, the code must be serialized and sent with the RMI call or with the mobile agent.

Another important aspect of remote configuration is security. Only authorized users should be allowed to use the external interfaces. This is not a problem because the M&M framework already has the necessary security features. It supports authentication and authorization of both mobile agents and RMI calls [17], so that when a request reaches the `ServiceManager`, it has already been authenticated. This allows the `ServiceManager` to grant or deny access based on the permissions associated with the authenticated user.

4 Related Work

There are some mobile agents systems with support for static extensibility, like the MOA platform [18] and Gypsy [19]. But to our knowledge, none of them supports dynamic reconfiguration.

Extensible architectures with support for dynamic reconfiguration is a topic that has been receiving increasing interest from the research community in the last few years.

dynamicTAO [20] is a reflective CORBA ORB, built on top of the TAO ORB. TAO is an extensible ORB, which uses the strategy pattern to provide static configuration.

Strategies are defined to encapsulate different aspects of the ORB internal engine. Configurations define which strategy will be used. At startup time, the configuration file is parsed and the TAO engine configures itself with those strategies. But once loaded, it is not possible to change the strategies. dynamicTAO extends TAO to allow on-the-fly reconfiguration. To accomplish this, it keeps a representation of the internal configuration (components and relations between them) and allows inspecting and changing that configuration.

The Distributed Multimedia Research Group at the Lancaster University has been working on a reflective architecture [21] based on the Python interpreted language. Python allows the programmer to inspect and change the implementation at run time. They take advantage of this feature to support adding or removing methods from objects and classes dynamically, and even change the class of an object at run time. This offers a very high level of dynamic reconfiguration.

COMERA [22] provides a framework based on Microsoft COM that allows users to modify several aspects of the communication middleware at run-time. It relies on the Custom Marshaler interface exported by COM, as well as the componentized architecture design that allows the use of user-specified components.

Although our work has some common ideas with these systems, it has a different focus. These systems focus on creating general architectures with support for dynamic reconfiguration, that can be later applied to different application domains. Our work focus specially on mobile agents systems, with the objective of creating an extensible middleware for building mobile agent enabled applications. Therefore, dynamic reconfiguration is just a tool to further improve the extensibility of the middleware.

5 Limitations and Future work

The main limitation of the approach presented in this paper is related to controlling access from clients to services. Our model only supports coarse-grained access control, in which a security verification is only made when a user first requests the service. Once the user is granted access to the service, he has full access to it. There should be support for fine-grained access control, because most services have methods with different levels of security requirements, so that some users might be granted access to some methods but not to others.

The problem with implementing fine-grained access control is that no one has all the knowledge necessary to define a fine-grained policy. The system administrator knows the users but might not know the service, and the service developer knows the service, but might not know the users. So even if a service developer uses the Java 2 security model, defining a set of permissions that protect the service, the system administrator might not know which permissions the service uses, and will not be able to map them to the users of its system.

A possible solution consists on using role-based access control [23]. A standard set of permissions, known both by the system administrator and by service developers, might represent different roles. A service developer would map these roles to concrete actions, and use them to protect the service methods. A system administrator would map the roles to users, giving each user a certain level of access. For instance, one role could represent read access. A user with that role would be able to execute the methods of all services that require only read access.

6 Conclusion

In this paper, we have described how dynamic reconfiguration can be implemented on mobile agent middleware. The middleware has an extensibility layer that supports plugging services, which are Javabeans components that provide new functionality to agents and to the application. The work presented here allows services to be plugged and unplugged at run time, without the need to shutdown the application. This allows building highly adaptable mobile agent platforms, whose functionality can be easily changed with minimal disruption to the service being provided.

Although the approach has been presented in the context of a particular mobile agent system, it can be applied to other mobile agent systems and, more generally, to any middleware system based on the Java platform that requires dynamic reconfiguration.

Acknowledgments

This investigation was partially supported by the Portuguese Research Agency FCT (M&M Project - reference POSI/33596/CHS1999 and Scholarship SFRH/BM/6787/2001), and by CISUC (R&D Unit 326/97).

References

1. C. Szyperski, *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, USA, 1998.
2. A. Carzaniga, G. P. Picco, and G. Vigna, "Designing distributed applications with a mobile code paradigm," in *Proceedings of the 19th International Conference on Software Engineering, Boston, MA, USA*, 1997.
3. P. Marques, L. Silva, and J. Gabriel, "Addressing the Question of Platform Extensibility in Mobile Agent Systems," in *Proceedings of International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000), ICSC Press, Wollongong, Australia*, December 2000.
4. Sun Microsystems Inc., "Java Platform." Available at: `http://java.sun.com/`.
5. J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., USA, 2000.
6. Sun Microsystems Inc., "JavaBeans Specification 1.01." Available at: `http://java.sun.com/beans/docs/spec.html`.

7. P. Marques, P. Santos, L. Silva, and J. G. Silva, "Supporting Disconnected Computing in Mobile Agent Systems," in *Proceedings of the 14th IASTED International Conference on Parallel and Distributed Computing and Systems (IASTED PDCS'2002)", IASTED/ACTA Press, Cambridge, USA*, November 2002.
8. Sun Microsystems Inc., "Java Remote Method Invocation Specification, JDK 1.3." Available at: `http://java.sun.com/products/jdk/rmi/`.
9. K. Moazami-Goudarzi, *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.
10. S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," in *Proceedings of Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98), Vancouver, British Columbia*, pp. 36–44, October 1998.
11. B. Venners, *Inside the (JAVA 2) Virtual Machine*. McGraw-Hill, second ed., 1999.
12. Sun Microsystems Inc., "The Java Virtual Machine Specification, Second Edition." Available at: `http://java.sun.com/docs/books/vmspec/`, 1997.
13. Sun Microsystems Inc., "Java Dynamic Proxy Classes." Available at: `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`.
14. M. Herlihy and B. Liskov, "A value transmission method for abstract data types," *ACM Transactions on Programming Languages and Systems*, pp. 527–551, 1982.
15. P. Marques, N. Santos, L. Silva, and J. Gabriel, "The Security Architecture of the M&M Mobile Agent Framework," in *Proceedings of SPIE's International Symposium on The Convergence of Information Tecnologies and Communications (ITCOM'2001), Proceedings of SPIE, Denver, Colorado, USA*, August 2001.
16. L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2," in *Proceedings of USENIX Symposium on Internet Technologies and Systems, Monterey, CA, USA*, pp. 103–112, 1997.
17. N. Santos, P. Marques, and L. Silva, "A Framework for Smart Proxies and Interceptors in RMI," in *Proceedings of ISCA 15th International Conference on Parallel and Distributed Computing Systems, Louisville, Kentucky, USA*, September 2002.
18. D. Milojicic, W. LaForge, and D. Chauhan, "Mobile objects and agents (MOA)," in *Proceedings of USENIX COOTS'98, Santa Fe, New Mexico, USA*, April 1998.
19. M. Jazayeri and W. Lugmayr, "Gypsy: A component-based mobile agent system," in *Proceedings of 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000), Rhodos, Greece*, 2000.
20. F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhães, and R. H. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000), New York, USA*, no. 1795 in LNCS, pp. 121–143, Springer-Verlag, April 2000.
21. N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair, "Towards a reflective component based middleware architecture," in *Proceedings of ECOOP'2000 Workshop on Reflection and Metalevel Architectures, Sophia Antipolis and Cannes, France*, June 2000.
22. W. Yi-Min and L. Woei-Jyh, "COMERA: COM extensible remoting architecture," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS), Santa Fe, New Mexico, USA*, USENIX, April 1998.
23. D. Ferraiolo and R. Kuhn, "Role-based access controls," in *Proceedings of 15th NIST-NCSC National Computer Security Conference, Baltimore, MD*, pp. 554–563, October 1992.