

# Dynamic Deployment of Services on Mobile Agents Systems

Nuno Santos, Paulo Marques, 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, allowing them to be used as reusable units of software. They have the added advantage of being able to migrate among 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, which is usually 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 implement 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]. It is structured as a set of JavaBeans [5] 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 `Services` are also JavaBeans components. After being plugged, services become available to any client of the extensibility layer, including the application, the mobile agents and other services.

Although this model has been used with success to implement and deploy several services, like agent tracking, inter-agent communication and disconnected computing [6], 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 sometimes is undesirable. This led us to improve the model, adding support for dynamic

reconfiguration.

This paper describes 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 [7] 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 provides an overview of related work. Finally, Section 4 concludes the paper.

## 2 Supporting Dynamic Services

### 2.1 Requirements

As identified in [8], there are three basic issues that must be addressed by any system that supports dynamic reconfiguration. They are **specification and management of change** (describing the changes to be made), **preservation of consistency** (the system should be working properly after reconfiguration), and **minimum disruption** (the application should continue to work during reconfiguration).

This work focuses on the last two issues, 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 support it at this moment. In future work that requirement will be addressed.

### 2.2 Java and Dynamic Reconfiguration

The Java platform has some very interesting features for implementing dynamic reconfiguration, like dynamic class loading [9] and reflection [4].

Class loaders are the entities responsible for loading and linking classes in the Java Virtual Machine, 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, giving 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 methods and fields, interfaces implemented and super-classes. This is important to support 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 a Service Descriptor, a Service Provider and a service interface together with its implementation.

A *Service Descriptor* is used to describe a service, defining its identity. It contains the name, version and the service interface name. Two services are compatible if they have the same name and service interface. This allows services to be upgraded, as it is explained below.

A *Service Provider* 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.

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, it is necessary to use a separate class loader for each service (Figure 1). 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.

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: when no more strong references [10] exist to it, the garbage collector can unload it. 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.

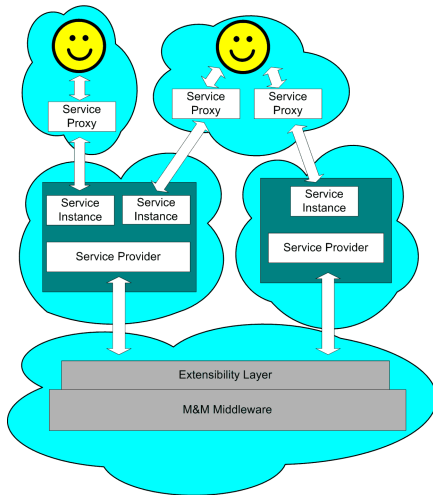


Figure 1: Dynamic Services. Each service is loaded by its own `ClassLoader`. The `ServiceManager` module can load and unload services.

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.

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 [10]. 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.

Direct references can be established in the two interfaces exported by services: 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, where there is a well-known abstract class from which the main class of applets and servlets must inherit. This abstract 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 instantiates 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's 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 in the client's class loader, and another in the service's class loader. This creates another problem. The service, when requested to create an 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** [11]. The middleware gives the client a dynamic proxy, which implements the interface class

defined in the client. Internally, the proxy's invocation uses reflection to invoke the service object. This avoids type-incompatibilities and direct references.

## 2.5 Upgrading Services

Loading, unloading and creating new instances of services is straightforward and therefore will not be explained here. But upgrading a service is a more elaborate procedure and deserves to be explained.

It would be possible to upgrade a service by first unloading the old version and then loading a new one. 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 [9]. Three issues must be taken care of. First, the state of the old class's objects must be transferred into objects of the new class. If the schema of the classes is different, it must also be adapted. Second, the static fields of the old class must be mapped into the new class. Finally, it must be ensured 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, if the schema of the classes is not the same or if the service has active threads or acquired locks, it is not clear how the middleware should perform the upgrade. 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 [12].

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.

## 2.6 Service Management

The previous discussion described how services are supported and managed internally by the middleware. It's still necessary to explain how services are viewed from

outside of the middleware, i.e., how services are managed and configured, how security is implemented, and how services interact with external entities. Unfortunately, due to space constraints, it is not possible to present a detailed discussion of these issues. Nevertheless, in this section we will briefly mention some of the more important ones.

The M&M middleware exposes the `ServiceManager` interface to allow the application to control the services. This interface allows services to be installed, removed, started, stopped, configured and instantiated, and is protected by fine-grained permissions, so that only authorized clients can access it.

This interface is available locally to the application. But it is also possible to access it remotely. The M&M middleware is capable of exporting a RMI interface that works as a gateway to the `ServiceManager`. Also, there is a service for agents which exposes the `ServiceManager` interface to them. This way, services can be deployed and configured remotely, using RMI or using mobile agents.

One other important issue is 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. To overcome this limitation two mechanisms are used. The first consists on using reflection to access the properties that are defined on the service provider of the service. The `ServiceManager` exposes the names of these properties, allowing the application to query and change them. The second way consists on using messages. The application can both broadcast and send messages to services. The `ServiceManager` forwards these messages to the target service providers, which can process or ignore them, as they seem fit. One example of this second mechanism is sending a shutdown message to all services.

## 3 Related Work

There are some mobile agents systems with support for static extensibility, like the MOA platform [13] and Gypsy [14]. 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. We will only mention briefly two of the most relevant projects.

dynamicTAO [15] is a reflective CORBA ORB, built on top of the TAO ORB. TAO is an extensible ORB with

support for static reconfiguration. dynamicTAO extends TAO to allow on-the-fly reconfiguration.

The Distributed Multimedia Research Group at the Lancaster University has been working on a reflective architecture [16] based on the Python interpreted language. This architecture supports adding or removing methods from objects and classes dynamically, and even change the class of an object at run time.

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.

## 4 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, providing new functionality to agents and to the application. The work presented here allows services to be plugged and unplugged at run time. 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 *Proc. of the 19th Int. 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 *Proc. of Int. ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizations and E-Commerce (MAMA'2000)*, ICSC Press, Wollongong, Australia, December 2000.
- [4] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., USA, 2000.
- [5] Sun Microsystems Inc., "JavaBeans Specification 1.01." Available at: <http://java.sun.com/beans/docs/spec.html>.
- [6] P. Marques, P. Santos, L. Silva, and J. G. Silva, "Supporting Disconnected Computing in Mobile Agent Systems," in *Proc. of the 14th IASTED Int. Conference on Parallel and Distributed Computing and Systems (PDCS'2002)*, Cambridge, USA, November 2002.
- [7] Sun Microsystems Inc., "Java Remote Method Invocation Specification, JDK 1.3." Available at: <http://java.sun.com/products/jdk/rmi/>.
- [8] K. Moazami-Goudarzi, *Consistency Preserving Dynamic Reconfiguration of Distributed Systems*. PhD thesis, Imperial College London, March 1999.
- [9] S. Liang and G. Bracha, "Dynamic class loading in the Java virtual machine," in *Proc. of Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, Vancouver, British Columbia, October 1998.
- [10] B. Venners, *Inside the (JAVA 2) Virtual Machine*. McGraw-Hill, second ed., 1999.
- [11] Sun Microsystems Inc., "Java Dynamic Proxy Classes." Available at: <http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html>.
- [12] M. Herlihy and B. Liskov, "A value transmission method for abstract data types," *ACM Transactions on Programming Languages and Systems*, 1982.
- [13] D. Milojevic, W. LaForge, and D. Chauhan, "Mobile objects and agents (MOA)," in *Proc. of USENIX COOTS'98, Santa Fe, New Mexico, USA, April 1998*.
- [14] M. Jazayeri and W. Lugmayr, "Gypsy: A component-based mobile agent system," in *Proc. of 8th Euromicro Workshop on Parallel and Distributed Processing (PDP2000)*, Rhodos, Greece, 2000.
- [15] 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 *Proc. of the IFIP/ACM Int. Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, New York, USA, April 2000.
- [16] N. Parlavantzas, G. Coulson, M. Clarke, and G. Blair, "Towards a reflective component based middleware architecture," in *Proc. of ECOOP'2000 Workshop on Reflection and Metalevel Architectures, Sophia Antipolis and Cannes, France, June 2000*.