Building a JMX management interface inside OSGi

Eric Fleury, Stéphane Frénot

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Building a JMX management interface inside OSGi

Eric Fleury — Stéphane Frénot

N° 5025

Decembre 2003

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--5025--FR+ENG

Building a JMX management interface inside OSGi

Eric Fleury* , Stéphane Frénot†

Thème 1 — Réseaux et systèmes
Projet ARÈS

Rapport de recherche n° 5025 — Decembre 2003 — 12 pages

Abstract: The "*pervasive computing*" will allow users to be able to utilize various applications through functional objects anytime and anywhere. Since pervasive computing is intrinsically highly dynamic and heterogeneous, applications should becomes "*context-dependent* " and able to change their functionality depending on dynamically changing user context. In a pervasive computing context, one must pay attention to the management part, inherent to the service deployment process. A automatic way to instrument and allow remote management control of distributed services becomes mandatory. This paper describes a supervising architecture for service oriented application in order to design and deploy context aware applications suitable to a ubiquitous Internet.

Key-words: JMX, OSGi, Service Oriented Programming (SOP), Management, Instrumentation, Supervision

* eric.fleury@@insa-lyon.fr
† stephane.frenot@insa-lyon.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Fabrication d'une interface de gestion JMX dans OSGi

Résumé : L'informatique pervasive permettera aux utilisateurs d'utiliser différentes applications n'importe où et n'importe quand. Du fait du caractère intrinsèquement dynamique et héterogène du matériel, les applications devraient être indépendantes du contexte et devraient pouvoir changer de fonctionnalités en fonction du contexte d'utilisation. Dans un contexte d'informatique pervasive, la question de la gestion des applications et un élément clé, directement lié au processus de déploiement de services. La possibilité d'instrumenter et d'administrer à distance les services distribués devient une nécessité. Ce papier décrit une architecture de supervision pour des applications orientées services afin de permettre la conception et le déploiement d'applications sensibles au contexte et donc adaptée au futur internet ubiquitaire.

Mots-clés : JMX, OSGi, Programmation Orientée Service (POS), Administration, Instrumentation, Supervision

1 Introduction

We believe that future Internet appliances will be small ubiquitous devices connected by different networks. The principal idea of pervasive networking is to change the role of communication networks. Currently, they are mostly used to interconnect static computers such as servers and workstations. Even if laptops or other portable devices are becoming wide spread, managing their mobility is still awkward and requires considerable user attention and effort.

Issues such as adaptation to the intrinsic dynamicity of such pervasive systems, context awareness, mobility and management are really challenging tasks once we try to build universal service framework. In order to design and deploy such kind of generic service framework [**?**, **?**, **?**], we must provide a light environment that will be based on a strong service oriented architectures that will be component oriented. Based on all these strong requirements, we find OSGi has a good candidate.

However, when designing and deploying dynamic context aware services in a distributed environment, even when relying upon advanced programming concept as SOP, one must pay attention to the management part, inherent to the service deployment process. A automatic way to instrument and allow remote management control of distributed services becomes mandatory.

In conjunction with the concept of service oriented architecture, we propose a complete management infrastructure for component oriented application. Our target is the management of distributed application deployed within a service oriented architecture, *i.e.*, the management of servers and of all underlying services. Thus, hot plug services can be deployed or redeployed dynamically. All management information attached to a service will also be automatically deployed for each new services. Inside each service, we need to systematically offer commands allowing the deployment, the configuration and the instrumentation in order to be able to administrate remotely every service. We will details in the section 2 our approach and its initial implementation based on OSGi and JMX.

This paper is organized as follow. The section 2 describes our approach based on the complementary use of OSGi and JMX. A discussion and plans for future works are presented in section 3.

2 JMX Management for OSGi platforms

In this section, we will present our integration of JMX inside OSGi. The main goal of this integration is to enable the management and instrumentation of OSGi components through a remote access provided by the JMX platform. We present the OSGi platform, the JMX platform and how we made the automatic instrumentation of OSGi services.

2.1 OSGi

The Open Service Gateway initiative (OSGi [**?**]) is an industry plan for a standard way to connect devices such as home appliances and security systems to the Internet. The "Service Gateway" is an application server in a computer that acts as gateway between the Internet and a home or small business's network of devices. The OSGi plans to specify the application programming interface for programmers to use and to allow communication and control between service providers and the devices within the home or small business. OSGi API is build on the Java programming language. In

this context OSGi provides a standardized infrastructure for Service Oriented Programming (SOP) on top of Java.

The OSGi specification defines the following elements:

The core framework The core framework is the central piece of the OSGi gateway. It is a daemon that guarantees the execution of the different hosted components. It authorizes and provides the bindings between clients requesting access to services and components implementing the corresponding service. In summary its main role is to record and to manage locally all the activity of the platform. From a Java point of view it can be viewed as a "super classloader" system that is able to manage class execution and association between components residing on the platform [**?**].

The standard services OSGi provides standard services for component execution, interaction and management. Theses services are only specified from an interface point of view. The standard services are: Package admin, Service tracker, Log Service, HTTP Service, Device Access, Configuration Admin, Metatype, Preference and User admin service.

A simplistic component model The OSGi component model relies on rather few concepts. The first one is the bundle concept. The bundle is the deployment unit. It is a jar resource file that can be installed on the framework. The bundle can specify several elements, thanks to a manifest file that describes it. The manifest file can declare services that the bundle offers, Java packages that the bundle makes available and finally it can declares natives libraries that can be used by Java classes.

From the component point of view, the core frameworks manages the life cycle of components. It controls whether the components are authorized to install, execute and exploit other component services. When component are deployed on the framework the manifest is read and service dependencies are controlled. If a service depends on a specific package provided by another service the frameworks make available that package to the requesting service. On the other side, if a bundle exporting a specific Java package is stopped, then the framework automatically stops all bundles that rely on that package.

Finally Java and OSGi provide a robust security mechanism for classes instantiation. The Java standard security manager is improved to integrate the life cycle management of OSGi components. Components can be downloaded from the Internet. The OSGi security manager enables a fine management of the components.

There are many implementation of the OSGi specification from many industrials (IBM's SMF [**?**], Sun's reference implementation JES [**?**], OpenSugar [**?**]). For our purpose we choose an open source OSGi implementation called Oscar [**?**]. Oscar is declared to be mostly compatible with OSGi specification.

In order to provide an external management layer to OSGi, we choose to implement JMX as an OSGi service. That service will provide a JMX view of the services managed by the OSGi core framework.

2.2 JMX: The management layer for OSGi

The Java Management Extensions Instrumentation and Agent [**?**] is an architecture, the design patterns, the APIs, and the services for application and network management and monitoring in the Java programming language. As for the OSGi specification the JMX architecture defines a component model managed by two layers: the agent and the instrumentation.

The agent layer This level represents the main JMX application (the MBean Server) which registers the different resources to manage and/or instrument. In JMX this is the main part of the specification; the agent manages all the resources and can gives access to them. The agent level on one side performs action on managed resources and on the other side gives an HTML/HTTP access to them. That remote access enables the resource management and instrumentation from any web client. Moreover the agent level, provides some standard services to management and instrumentation aware applications. Those services enable the notification of new administrable resources (through a notification mechanism), they enable the positioning of counters, timer and gauges that automatically instrument the resource and triggers notifications on some events.

The instrumentation layer The instrumentation layer is responsible of interactions with resources. At that level resources are modeled as MBeans. MBeans are JMX components that wrap access to the resources. They provide a Java interface that declares methods that can be used for management and instrumentation of the resource. Each managed resource is seen as a component managed by the agent level. JMX defines different kinds of MBeans (standard, dynamic, open and model) that can model in different ways the resources.

The JMX architecture enables the instrumentation and management of any kind of application, provided one can make a MBean component for its management.

2.3 JMX dynamic instrumentation of OSGi services

Both OSGi and JMX are rather similar since they both rely on a framework for component execution (the bundlecontext for OSGi, the Agent layer for JMX), they both provide standard services for component management and finally they both provide some kind of simple component model suitable for application development. Nevertheless we consider these two platforms as complementary rather than concurrent. For OSGi its main advantages are that it can manage complex components dependencies, it has a more complete component life cycle model, and finally the fact that the security model is augmented from Java. On the other side JMX enables remote interaction with components through HTTP/XML access and it enables easy integration of instrumentation capabilities on MBeans. We decide to have OSGi as the main execution platform for component and bundelize JMX in order to get its facilities. This architecture is shown in Fig. 1

In this figure we can see that every service is packaged as an OSGi service. In particular JMX is just another OSGi service. The JMX service manages internally MBeans that represent other OSGi services.

In our approach we want to give access to OSGi services through JMX. The main steps to achieve this are:

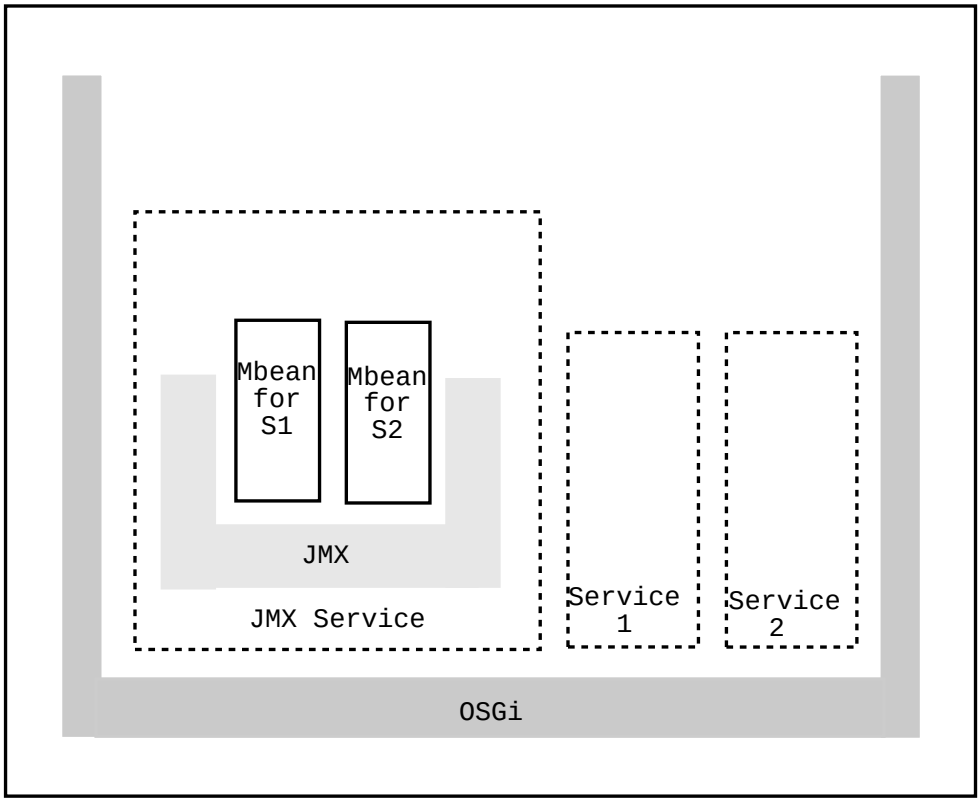


Figure 1: JMX inside OSGi main view

bundelize JMX:

In this step we need to launch a JMX platform from OSGi. We have put MX4J (an open source JMX implementation [**?**]) in an OSGi bundle. This bundle makes available the JMX packages and an OSGi service that can be used to dialog with the JMX agent layer.

MBeanify Running OSGi Services:

In this step we query OSGi in order to get all registered services and generate a DynamicMBean for each one.

Fig. 2 represents services involved in the beanification process.

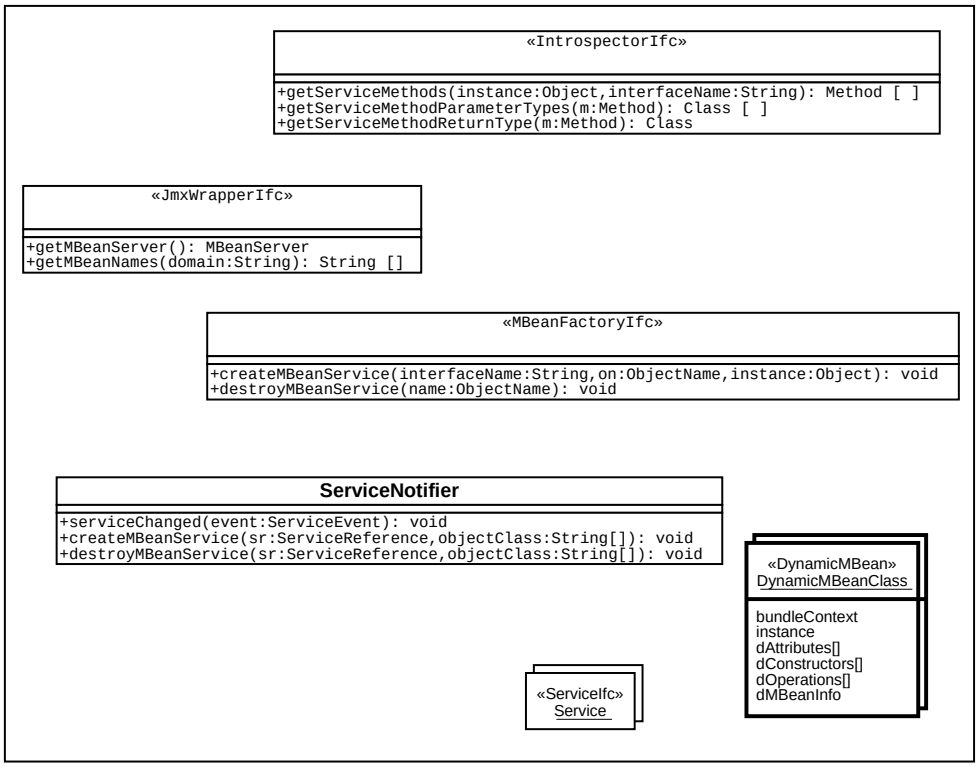


Figure 2: Services

The system relies on four services.

- The "JmxWrapper Service"

  It wraps the original JMX service into an OSGi bundle. The bundle embeds the original MX4J jar, and provides access to the MBeanServer singleton. External services can request that service in order to get all registered MBeans.

- The "MBeanFactory"

  That service extracts the OSGi service interface, builds a dynamic MBean that acts as a proxy for the OSGi service and finally register the MBean at the MBeanServer (through the JmxWrapper Service).

- The "Introspector"

  The service is used by the MBeanFactory in order to extract the methods from the OSGi service interface.

- The "Service Notifier"

  It is not a service, it listens to service registration/unregistration on the OSGi framework and asks the MBeanFactory create/destroy the corresponding MBean. Fig. 3 illustrates the sequence diagram that occurs when starting the Notifier service.

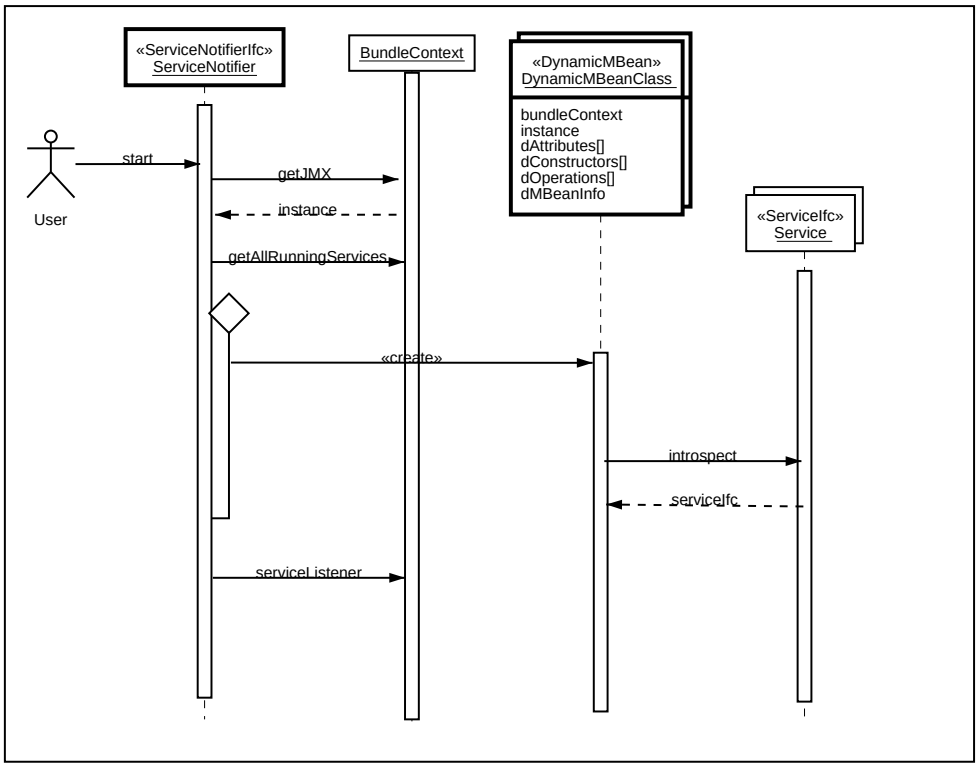


Figure 3: UML sequence diagram

When the user starts the notifier, it looks for the JMX agent service (the MBean server). Then for each service registered in the framework, the notifier service asks for the creation of a wrapping DynamicMBean. Then, each service becomes available for JMX management.

When the notifier is started, every OSGi service becomes available through JMX management. For instance Fig. 4 represents the main JMX access to OSGi services.

We have defined two JMX naming domains. The "`Domain:Service`" represents the services that generates the dynamic MBeans. The "`Domain:OsgiOscarService`" represents the dynamically instrumented OSGi services. The two remaining JMX Domains ("`Domain:HtmlAdaptor`" and "`Domain:JMImplementation`" are domains related to MX4J implementation.

In order to manage the OSGi shell we have developed a simple JMXShell service that provides all commands to manipulate Oscar's shell. We can notice in the previous figure (Fig. 4) the last line that gives access to that service. If we click on the "`Service:name= shellmbean.ShellMbeanService@861f24`" hyper link, we get the following web page (Fig. 5).

Through that web page, we can manipulate the Oscar shell. Every methods of the OSGi service interface is available inside the HTML form. In order to invoke the "`ps -l`" command, we need to type the "`-l`" option in the textfield area and click on the "invoke" button (not visible on the Figure). If we invoke the "`ps -l`" Oscar command, we obtain the next web page (see Fig 6).

The "ps -l" command displays all bundles currently deployed on the OSGi gateway.

Our integration of JMX inside OSGi enables us to manipulate every OSGi platform and its services through a simple URL access. Every interaction is made with a web navigator that interacts with the JMX bundle provided by the OSGi platform. All our bundles and developments are open source and are available on `http://ares.insa-lyon.fr/~dan/jmx_osgi/`

3 Conclusion

We have presented the implementation of an automatic way to instrument and allow remote management control of services within a service oriented architecture. Our implementation is based on the complementary use of OSGi and JMX. Indeed, our integration of JMX inside OSGi enables the management and the instrumentation of OSGi components through a remote access provided by the JMX platform.

From a performance point of view, the MBean indirection cost is only paid when there is a management and/or instrumentation operation. In the standard execution scheme, calls are not going through the Mbeans.

Possible extension of this work is open to investigation. We are currently working on taking into account the last release of OSGi that offers several improvements. However, the version 3 of OSGi still does not consider remote service management. Our approach offers a large flexibility and allows us to deploy an instrumentation framework of distributed services applications. Our continuing work in this area is to design a more ambitious framework and to propose a *peerware* that will enable to connect several OSGi platforms. Using such a paradigm, our approach still provides a global remote management of all OSGi platforms. The underlying idea that we are currently implementing is to use a middleware concept to exchange information based on a peer-to-peer approach. In order

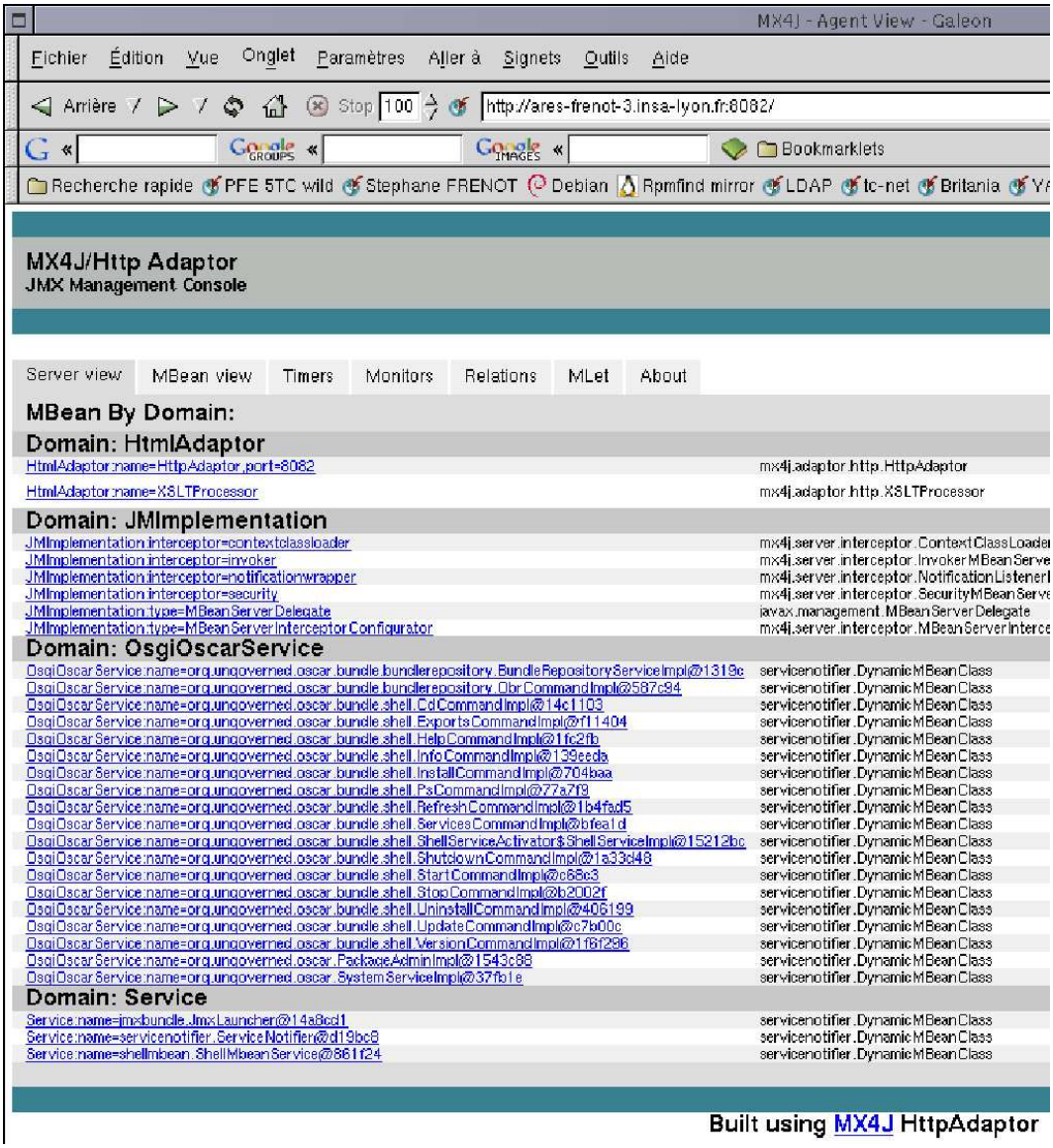
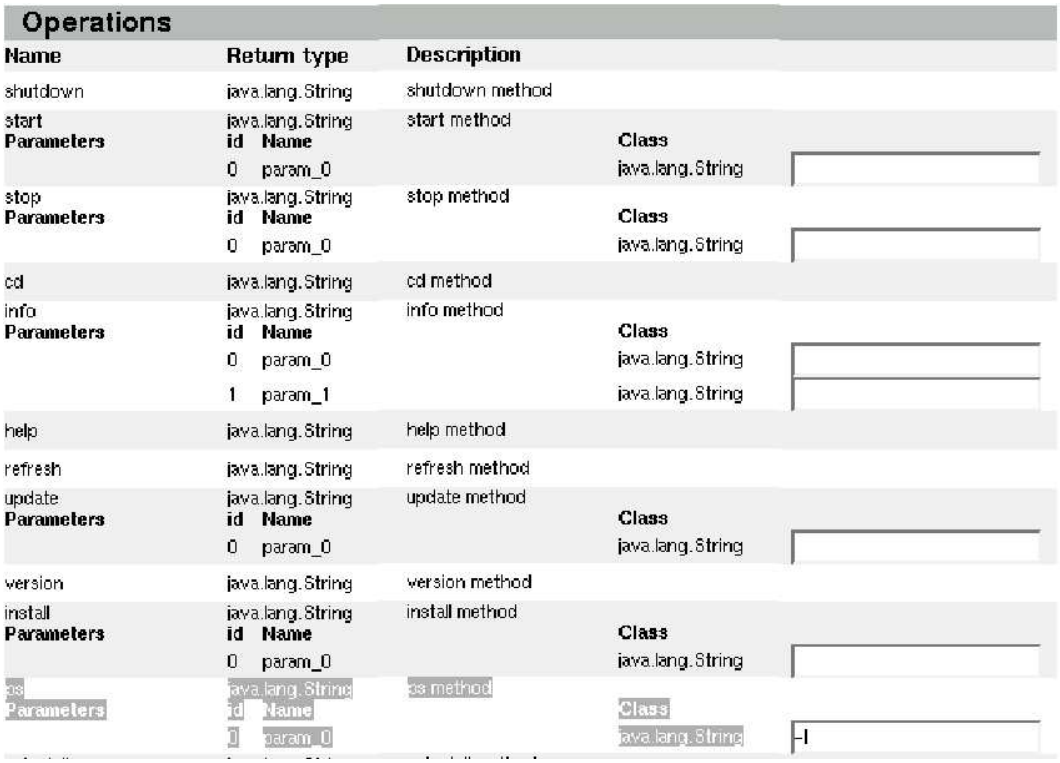


Figure 4: JMX main page



Operations

Name	Return type	Description		
shutdown	java.lang.String	shutdown method		
start	java.lang.String	start method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	
stop	java.lang.String	stop method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	
cd	java.lang.String	cd method		
info	java.lang.String	info method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	
	1 param_1		java.lang.String	
help	java.lang.String	help method		
refresh	java.lang.String	refresh method		
update	java.lang.String	update method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	
version	java.lang.String	version method		
install	java.lang.String	install method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	
ps	java.lang.String	ps method		
Parameters	**id Name**		**Class**	
	0 param_0		java.lang.String	-l

Figure 5: Access to one OSGi service

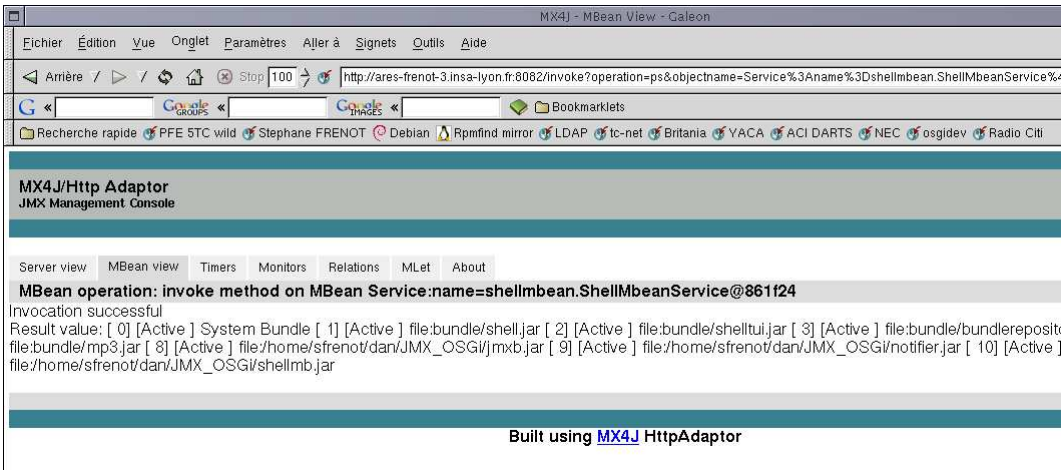


Figure 6: Execution of the ps command

to inter connect several OSGi platforms together and still taking benefit from the Java advantage (portability, safety and security features, increased productivity) we plan to use JXTA [**?**]. Based on this well defined paradigms and clearly identified tools, we are close to the main goal stated in the introduction.

We are also working on improvements of our integration of JMX inside OSGi. Our first implementation is a coarse grain service since we choose to implement JMX as an OSGi wrapper service. A finer grain integration is under consideration to reduce the overall size by deleting redundant services. For instance, a service such as HTTP is provided by both JMX/MX4J and OSGi/Oscar. This more integrated plateform should have better performances than the current coarse grain implementation.

The last ongoing work is to integrate the CIM Application Management Model which is an information model that describes the details commonly required to manage software products and applications. We want to integrate the CIM model into our approach in order to use more sophisticated management platform instead of the basic JMX HTTP adapter. Note that the aim of the JSR 146 *WBEM Services: JMX Provider Protocol Adapter* is to define how JMX instrumentation can be mapped to CIM and the definition of a JMX Provider Protocol Adapter for WBEM Services.

Finally, open questions remains in the management area of service oriented application. What kind of information provided by the middleware are worth exploitable and useful for the application? How a distributed service can provide its own supervising characteristics and integrates itself automatically in existing supervising applications?

Unité de recherche INRIA Rhône-Alpes
655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifi que
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399