J2EE packaging, deployment and reconfiguration using a general component model

Takoua Abdellatif[1,2], Jakub Kornaś[2], and Jean-Bernard Stefani[2]

[1] Bull SA
[2] LSR-IMAG laboratory (CNRS, INPG, UJF) - INRIA - Sardes project
INRIA Rhône-Alpes, 655 av. de l'Europe, F-38334 Saint-Ismier Cedex, France
Firstname.Lastname@inrialpes.fr

Abstract. This paper describes a case study of enhancing the deployment process in J2EE application servers (AS), and more precisely the services building such servers and the applications executing on the servers. We show how, by following a component-based approach to the design of the server, we address the versioning and licensing issues raised by the fact that a J2EE server is built out of heterogeneous, third-party software.
As a proof of concept, we present a re-engineered version of the JOnAS J2EE server implemented using Fractal, a component model providing flexible control capabilities and hierarchical composition. We describe how Fractal packaging together with a JOnAS-specific deployment system are used to deploy and reconfigure our Fractal-based version of the JOnAS server. Finally, we show how the same model and packaging can be used to deploy applications executing on the server.

1 Introduction

J2EE [1] application servers are complex, service-oriented architectures. Existing open-source solutions usually implement services as wrappers of legacy code. For example, the JOnAS [2] application server[3] contains a Web service wrapping either Tomcat [3] or Jetty [4], a transaction service that wraps JOTM [5], etc. Since services are developed by third parties, it is necessary to allow deploying and updating them independently. Indeed, the application server must both allow choosing between different licenses at deployment time, and allow services to be updated when a new version is available.

Service update issues are not handled by current J2EE specifications. The JSR88 [6] focuses on the deployment of applications, but not on the deployment of the middleware. JSR77 [7] defines an information model that must be exposed to managers in charge of monitoring and controlling the system. This model does not contain the necessary information for dynamically updating services. This lack of specification regarding the middleware management (e.g. licensing, versioning) is currently left to the server providers.

[3] JOnAS is an open-source application server freely available under an LGPL license at http://jonas.objectweb.org

Updating a service code at runtime (i.e. without stopping the server) involves three main tasks: (i) isolation of services as independent packages, (ii) deployment and redeployment of services, and (iii) handling service dependencies and state at runtime. Regarding the packaging part, the main issue is to handle the dependency between the different packages and the compatibility between code versions they contain. Regarding the deployment and redeployment part, each service needs to have an independent life cycle, in particular it must be deployable independently from other services. Finally, handling the running service state implies taking into account the service's stateful data and dependencies between the updated service and other running services.

Issues raised in point (iii) necessitate reconsidering the middleware architecture. Indeed, service dependencies must be explicit so that when a service is updated, the behavior of dependent services can be controlled. This has been the purpose of previous work [8] on JOnAS, which led to the JonasALaCarte prototype. JonasALaCarte adopts a component-based architecture, implemented using the Fractal [9] component model. Fractal allows building hierarchical architectures (using composite components), where components communicate through explicit bindings. JonasALaCarte uses Fractal components to wrap services, thus making them independent units of configuration and deployment with an independent lifecycle.

This paper focuses on points (i) and (ii), i.e. the packaging and deployment parts. We show that we can adopt the same component model (Fractal) to implement the service packaging and the deployment infrastructure. Service packages are represented by Fractal components and dependencies between packages are expressed using Fractal bindings. Moreover, the deployment infrastructure is implemented using Fractal components, which allows dynamically plugging various deployment policies adapted to the deployment environment (e.g. centralized, clusters, grids, etc.).

Furthermore, we show that our deployment tool and packaging model is also applicable to J2EE applications. Unifying the packaging and deployment process thanks to the Fractal component model allows for abstracting the management tasks (packaging, deployment and system adaptation) to a configuration of Fractal components. Regarding existent solutions in open source application servers, our management solution is uniform: we use the same model for the packaging and the AS execution at runtime. Moreover, we adopt the same package structure for the middleware services and the J2EE applications.

The main innovative aspects of our work are that: we allow for versioning and redeployment of services building the JOnAS J2EE server, we solve the possible licensing issues by packaging each JOnAS service independently, we use a uniform component model from the package level, through the AS level, to the application level, and finally, in our solution, dependencies between JOnAS services are made explicit and map on package dependencies.

The rest of the paper is structured as follows: in section 2 we briefly introduce the Fractal component model and Fractal packaging. Section 3 describes the drawbacks of JOnAS in terms of deployment and presents how, by re-engineering

JOnAS, we have obtained JonasALaCarte, a Fractal-based version of the server. Sections 4 and 5 present how components building the JonasALaCarte are packaged, deployed and redeployed. In section 6 we describe the related work before concluding the article in section 7.

2 Fractal component model

In this section we briefly describe the Fractal component model: the principles underlying the model, the Fractal ADL (architecture description language) and Fractal packaging.

2.1 Fractal principles

Fractal [10] is a general component model. It distinguishes two types of components: *primitive* and *composite*. Primitive components are standard Java classes that conform to certain coding conventions. Composite components encapsulate a group of primitive and/or composite components.

A Fractal component is made of two parts: a *controller part*, which exposes the component's interfaces and comprises controller and interceptor objects, and a *content part*, which can be either a standard Java class in case of a primitive component, or other components (called subcomponents), in case of a composite component.

Similar to other component models, Fractal distinguishes *server interfaces*, which correspond to provided services, and *client interfaces*, which correspond to required services. Moreover, Fractal supports both primitive bindings (i.e. Java references) and composite bindings which are built out of a set of primitive bindings and binding components (stubs, skeletons, adapters, etc).

Figure 1 illustrates the different constructs in a typical Fractal component architecture. The gray boxes denote the controller part of the components. Arrows correspond to bindings; the interfaces appearing on the top of a component represent the controllers, the interfaces on the left are server interfaces and on the right are client interfaces.

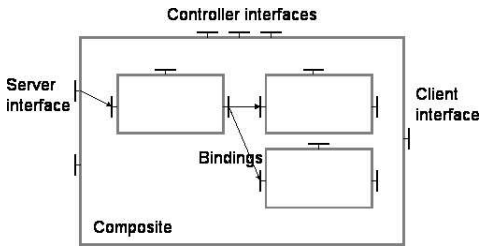

Fig. 1. An example Fractal architecture.

The construction of a system with Fractal component yields a dynamically adaptable system where the component is the unit of configuration, deployment and reconfiguration. The system architecture, written in ADL, is expressed in terms of the component model, exhibiting bindings between components and containment relationships. These properties are specific to Fractal, compared to other component models, as explained more in detail in [10]. For these reasons we chose to build our new AS, as well as the deployment system itself, using Fractal.

2.2 Fractal ADL

Fractal Architecture Description Language (ADL) is a mean to define architectures of Fractal applications. It is XML-based, and each description of the Fractal architecture is stored in a `.fractal` file. A sample ADL description of a Fractal component is represented in figure 2. This example corresponds to an ADL description of a component named `WebContainer`. This component has four client (required) interfaces, named `jmx`, `security`, `jprop` and `lmgr`, implemented by the following Java interfaces: `JmxServiceItf`, `SecurityServiceItf`, `JProperty` and `LManager`. The `WebContainer` component also has one server (provided) interface, named `service` and implemented by a Java interface `ServiceItf`. Finally, the implementation of this component's functional (content) part is provided by a Java class called `WebContainerWrapper`. `WebContainer` can be bound to other components, it can also be a subcomponent of some other component.

```
<definition name="WebContainer">
  <interface name="jmx" role="client" signature="JmxServiceItf"/>
  <interface name="security" role="client" signature="SecurityServiceItf"/>
  <interface name="jprop" role="client" signature="JProperty"/>
  <interface name="lmgr" role="client" signature="LManager"/>
  <interface name="service" role="server" signature="ServiceItf"/>
  <content class="WebContainerWrapper"/>
</definition>
```

Fig. 2. `WebContainer.fractal`: A sample ADL description of a Fractal component

Fractal ADL has been designed to be open and extensible: it is made of several units, where each unit defines syntax for one architectural aspect (like interface, binding, attribute etc). Developers are free to define their own units. At deployment time, an ADL description of the application is parsed by a factory tool. This factory tool can also be extended to take into account added units.

2.3 Fractal packaging

Fractal packages are used to deploy Fractal applications. These packages are stored in package repositories, which can be of various kinds, such as file systems, databases etc. Each Fractal package `A.far` is a Fractal component `A` in

a serialized form, which is described by a Fractal ADL definition `A.fractal` contained in the package itself. Such a definition of packages unifies the package and component concepts, in the sense that a package is just a special form of a component. All the properties of Fractal packages are deduced from this identity relation: for example, Fractal packages can contain sub-packages, just like Fractal components can contain subcomponents. Figure 3 illustrates an example of Fractal packages, including their metadata.

```
package JonasALaCArte.far
   JonasALaCarte.fractal:
     <definition name="JonasALaCarte" version="1.0">
       <component name="WebContainer" definition="WebContainer"/>
       ...
     </definition>
   ow_jonasbootstrap.jar
   ... other, non ADL files ...

package WebContainer.far
   WebContainer.fractal:
     <definition name="WebContainer" version="1.0">
       <interface name="jmx" role="client"
         signature="JmxServiceItf"/>
       <interface name="security" role="client"
         signature="SecurityServiceItf"/>
       ...
     </definition>
   catalina.jar
   tomcat-coyote.jar
   ... other, non ADL files ...

package JMX.far
   JMX.fractal:
     <definition name="JMX" version="1.0">
       <interface name="service" role="server"
         signature="JmxServiceItf"/>
       ...
     </definition>
   jmx.jar
   ... other, non ADL files ...
```

Fig. 3. An example of Fractal package files

Dependencies between Fractal components can only be of two sorts: dependencies through component encapsulation, and dependencies through component interfaces. These two dependency types give two dependency types between Fractal packages : (i) a containment dependency gives a strong dependency between two packages (in the example presented in figure 3, the containment of `WebContainer` inside `JonasALaCarte`, gives a strong dependency between package `JonasALaCarte.far` and package `WebContainer.far`) (ii) a dependency through interfaces gives a loose dependency between packages (in the example presented in figure 3, the client interface `jmx` gives a loose dependency between package `WebContainer.far` and any package that provides the `JmxServiceItf` interface), which in the example presented in figure 3 is the `JMX.far` package.

3 Re-engineering JOnAS using Fractal

In this section we first briefly describe the JOnAS J2EE server. Then we outline the drawbacks of the existing implementation of JOnAS focusing mainly on deployment, licensing and updates issues. Finally we present our re-engineered, Fractal-based version of the server which we call JonasALaCarte. In the next two sections we explain how this re-engineering work allows us to address all the deployment-related issues of the "standard" server.

3.1 The JOnAS server

JOnAS is an open source J2EE application server. It is developed within the ObjectWeb Consortium [11]. The server's role is to host J2EE-compliant applications by providing them with an execution environment that offers a well-defined set of non-functional services (persistency, transactions, security, etc.). To achieve that, the server integrates various software from different providers, such as the Apache Software Foundation [12], the ObjectWeb consortium etc. This heterogeneous software builds the services that offer non-functional properties to the J2EE applications. Even though each software providing different non-functional aspect could be considered as an independent component with explicitly defined relations to other components, JOnAS does not employ a component-based approach in its design. On the contrary, JOnAS is a monolithic block of code in the sense that the relations between the third-party "components" integrated by the server are not explicit - they are hard-coded in JOnAS' classes. Such an approach has major drawbacks in terms of both architecture and deployment. In terms of software architecture, the non-component-based approach makes the internals of JOnAS difficult to understand and the server difficult to manage at runtime. In terms of deployment, it does not allow the redeployment of only parts of the server - since JOnAS services are not components, they cannot have a life cycle independent of the life cycle of the server. Therefore, it is impossible to, for example, redeploy JOnAS services independently. Moreover, it is impossible to address the licensing issues raised by the fact that for certain third-party components it can be illegal to package and distribute them together with other third-party components. To address these issues we have re-engineered the JOnAS server to obtain a component-based version of it.

3.2 Fractalized JOnAS

JonasALaCarte [8] is our re-engineered, component-based version of the JOnAS server. In our re-engineering work we have adopted the Fractal component model.

As a result of our re-engineering work, all JOnAS services and management entities became Fractal components. As illustrated in figure 4, each instance of the JonasALaCarte server is therefore a composite component encapsulating a set of interacting services (primitive components). The latter are bound using Fractal bindings. The first advantage of such an approach, compared to

traditional JOnAS server, is that the architecture of the server is explicit - connections between services are well defined, services building the server can be managed thanks to the control interfaces of the components that wrap these services. Second advantage is that components building the server can be packaged, deployed and redeployed independently. Note, however, that for most of these components it is impossible to have two versions of them running in a single application server. This is due to the way these components are implemented.

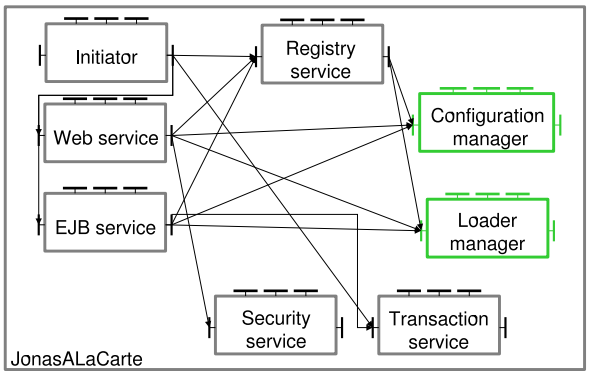

Fig. 4. Architecture of the JonasALaCarte J2EE server.

We have implemented a JSR77 lifecycle controller of each deployable Fractal component (in our case the services and the management components) as a Fractal controller. Figure 5 presents the JSR77 automate. When a Fractal component is deployed, its state is set to `Starting`. Each service component asks the `Loader Manager` for its class loader, performs some initialization operations and starts. If the service starting succeeds, its lifecycle controller is positioned to the `Running` state; otherwise the latter is set to `Failed`. To stop a service its state is set to `Stopping` and the service performs some state storage and clean-up. If these operations succeed, the service sets its state to `Stopped`, otherwise it is set within a configurable time-out to `Failed`.

4 Packaging

In this section we describe how Fractal packaging applies in the context of the Fractalized JOnAS server.

Fractalized JOnAS services are fractal components, therefore packages used for storage and deployment of these services are Fractal packages (serialized forms of Fractal components).

As stated in section 2.3, package-level dependencies between services building the server are the same as runtime-level dependencies, and are therefore

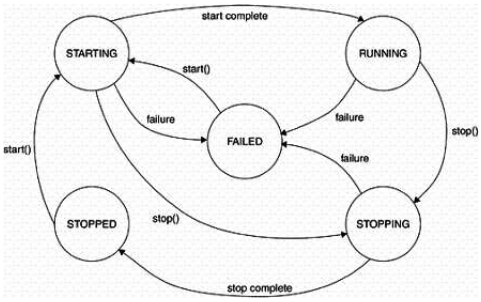

Fig. 5. JSR77 component lifecycle automate.

expressed in the same ADL file. However, this file is enriched with versioning information needed by the package-management system. This information is used to solve package dependency and compatibility issues.

In addition to the `.fractal` file, each package contains also the `.jar` files that provide the actual code needed by the services at execution.

```
package WebContainer-1.0.far
   WebContainer.fractal:
     <definition name="org.objectweb.jonasALaCarte.WebContainer"
       version="1.0">
       <interface name="jmx" role="client"
         signature="org.objectweb.jonas.jmx.JmxServiceItf"
         compatibility="[1.0, *]"/>
       <interface name="security" role="client"
         signature="org.objectweb.jonas.security.SecurityServiceItf"
         compatibility="[1.0, *]"/>
       <interface name="jprop" role="client"
         signature="org.objectweb.jonasALaCarte.configurator.JProperty"
         compatibility="[1.0, *]"/>
       <interface name="lmgr" role="client"
         signature="org.objectweb.jonasALaCarte.loaderManager.LManager"
         compatibility="[1.0, *]"/>
       <interface name="service" role="server"
         signature="org.objectweb.jonas.service.ServiceItf"/>
       <content
       class="org.objectweb.jonas.web.wrapper.catalina55.WebContainerWrapper"/>
       ... the rest of the ADL file
     </definition>
   catalina.jar
   tomcat-coyote.jar
   ... other, non ADL files ...
```

Fig. 6. The content, including metadata, of the `WebContainer-1.0.far` package

As can be seen, dependencies between packages correspond to the runtime dependencies between services building the server. Therefore, the `WebContainer-1.0.far` package depends on any package providing the `JmxServiceItf` interface, any package providing the `SecurityServiceItf` in-

terface etc. These package dependencies are resolved by the JonasALaCarte deployment mechanism.

The J2EE modules (WARs, EARs, RARs and EJBJars), as defined in JSR88 are wrapped as Fractal components. The ADL files in the `.far` archives describe the module version and the dependencies between the module and the services where it will be deployed. The deployment manager checks the code version compatibility. On the other hand, we express in ADLs, the dependencies between the modules themselves. For example, it is possible that two EARs archives need to share the same RARs. Note that JSR88 specification does not address the dependency between modules. We offer this feature thanks to our packaging structure without breaking the specification. Again, adopting the same package structure for both applications and middleware allows using the same APIs and management tools.

Since packages are only units of code distribution, they do not provide information on how the code contained in a package should be loaded at runtime, that is information relative to class loading. This is important in the context of JOnAS, since the server employs a rather complex class-loading hierarchy, allowing for example for run-time versioning of code. We believe that this class loading hierarchy is orthogonal to the code packaging - it is the deployment tool's role to have enough knowledge and means to create class loader hierarchies for the packages it obtains from package repositories. Thus, as will be explained in the next section, our deployment tool creates a proper class loader hierarchy for JOnAS.

5 Deployment and Updates

This section describes how Fractal packages containing the JOnAS services are deployed by our deployment tool and how JOnAS services can be redeployed without the need to redeploy the whole server. The first part of this section describes the architecture of the JonasALaCarte deployment tool, the second part outlines the properties of this deployment tool and finally the third part presents a redeployment use case.

5.1 Architecture

As illustrated in figure 7, our deployment tool is built of the following components: The `Deployment Manager`, the `Repository` and the `Loader Manager`. The role of the `Deployment Manager` is to parse the package meta-data, to identify the dependencies between the packages and check package availability and version compatibility. If no problem (lack of necessary packages, package incompatibility, etc.) is detected, the deployment manager extracts the content of `.far` packages (middleware or application ones) and asks the `Repository` component to store locally the content of these packages. Various storage policies can be implemented. In our current implementation, the `Repository` component stores the contents of packages in a folder structure equivalent to the one defined by

JOnAS. It is possible to have other storage semantics or other storage support like data bases for persistence. The deployment manager asks the `Initiator` component to deploy services. The latter invokes then the *start* interfaces on the different service components.

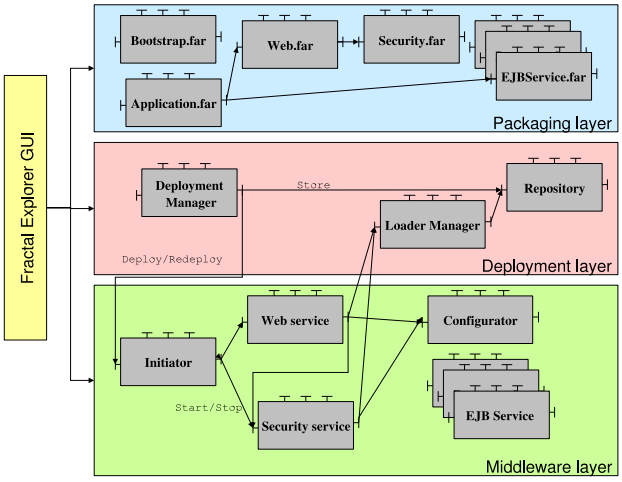

Fig. 7. Uniform Fractal-based representation of packages, the deployment infrastructure and the middleware

The `Initiator` component polls the service lifecycle states and sends a notification to the deployment manager if a service deployment fails. As a future work we plan to implement some repair deployment policies as Fractal components. Note that currently the same protocol is used for the deployment of J2EE applications [13].

5.2 Properties

Our deployment architecture has three main properties:

Separation of deployment policies from deployment mechanisms by adopting a component-based architecture. In our context, the deployment policies define (i) how packages are stored, (ii) how dependencies between packages are handled, and (iii) how class loaders are created. Implementing these three concerns using separate components communicating through well defined interfaces allows modifying the behavior of each component independently from the others. For example, our existing implementation of the `Loader Manager` component creates a class loader hierarchy equivalent to the one used in JOnAS [14]. Another possible implementation could use Module Loader [15] as a class loading mechanism, and thus allow for the usage of any of the module loader's search

policies. Finally, an implementation of the `Loader Manager` could map directly the packaged components, based on their metadata, on the namespaces provided by class loaders as we explain it in [16].

Unification of the packaging and the deployment tools by adopting the component model used to build the middleware itself. Adopting the same component model allows abstracting the management of the different phases to the configuration of Fractal components. This unification allows for using the same API to monitor and manage the application server lifecycle steps: from packaging to runtime execution. Currently, we enhanced the fractal explorer tool [17] to offer a common GUI for these steps. Figure 8 illustrates the navigation through Fractal packages of the server, the deployment components and the middleware services. Later in this section we explain in detail how this GUI tool allows for management and redeployment of components building the JonasALaCarte server.

Unification of the packaging and the deployment of J2EE applications and middleware. To achieve this goal, we adopt the same package structure and deployment tool for both middleware and applications. Indeed, the middleware services as well as J2EE applications are packaged as Fractal packages (`.fars`).

5.3 Redeployment use case

The `Deployment Manager` allows for redeployment of services. For that it obtains the new `far` from the package repository, asks the `Initiator` to store the `far`'s content locally and calls the *deploy* interface of the `Initiator` component. The `Initiator` calls *redeploy* interface of the service. The service is then stopped together with all the services that use it. The service component subject to redeployment asks the `Loader Manager` for its new class loader and restarts.

The redeployment of JonasALaCarte services can be performed using the Fractal Explorer tool that we have extended for our needs. Figure 8 illustrates an example of redeploying the `WebContainer` component. Figure 8a presents the initial state of the redeployment operation. On the left we can see all the subcomponents of the JonasALaCarte composite component, including the `WebContainer` component. We can also see the `redeployment-controller`, a Fractal controller specific to JonasALaCarte, responsible for initiating the redeployment of services. In figure 8b we see that the administrator chose the `WebContainer` component for redeployment. At this stage, the deployment manager queries the (possibly remote) package repository for the available versions of `.far` files containing the `WebContainer` component. As can be seen in figure 8c, two versions of this component are available and the administrator decides to deploy the "2.0" version of the component. This example shows how we achieve our goal of being capable to redeploy the services building the JOnAS server. Moreover, since services are packaged independently, we also solve the possible licensing issues. It has to be mentioned that redeployment of the `WebContainer` also involves the redeployment of all `war` and `ear` files.

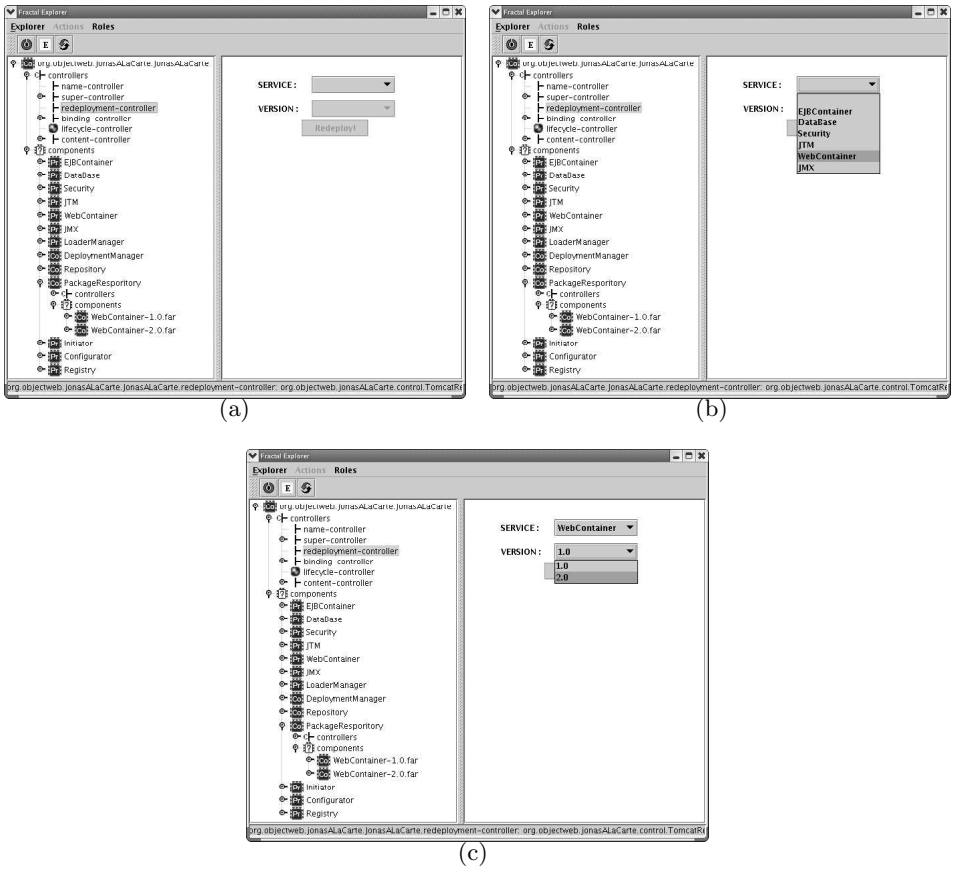

Fig. 8. Example of JonasALaCarte service redeployment using Fractal Explorer titi

In the use case described above we do not address the state preservation issue. The capacity to provide such state preservation depends on the properties of legacy software building the server.

6 Related work

The related work can be divided into two types of systems: the J2EE application servers and the generic, "Module" systems for Java.

6.1 J2EE servers

In JBoss [18], the architecture of the AS is completely based on JMX. A JMX agent represents the middleware kernel. The services are implemented as MBeans and are deployed using the MLet service. In JBoss, some tools are built on top of

JMX to express the dependencies between MBeans and thus between services. JBoss adopts different packaging structures for the application deployment modules (JARs, RARs, EARs) and the service deployment modules (SARs). Consequently, there is no uniform package structure for the middleware and the application parts. MBeans architecture unit is not exploited for the packaging and the deployment. Regarding JBoss, JonasALaCarte is based on a uniform model, the Fractal component model, for the packaging, the deployment and the middleware architecture. Furthermore, we are not aware of the dynamic versioning feature in JBoss, which we provide in JonasALaCarte.

Geronimo [19] developers are aware of the JMX limitation to uniform the complexity of the system. In fact, relation betweens the MBeans and the communication between them are not in the scope of the JMX model. Geronimo adopts instead an IoC [20] kernel based architecture. Inversion of Control, also called dependency injection, is a pattern supported by IoC containers and frameworks to achieve separation of concerns. Components inside the container can isolate dependencies and have these dependencies injected into them during execution/deployment. Components inside Geronimo are called GBeans and are the manageable units in Geronimo. The deployment process is separated into the user part (the modules creation) and the execution part (the configurations creation). During deployment, one or more modules are packaged together into a configuration. Internally, Geronimo sees only configurations, as packaged deployment of one or more GBeans. Like Fractal ADLs in JonasALaCarte, a Geronimo deployment plan is the Geronimo-specific meta-data. Like in Geronimo, our packaging and deployment tools aim at unifying the complex system by adopting the same structure of the deployment modules and deployment plans for the middleware and the J2EE applications. However, in JonasALaCarte, we adopt the same model for the packaging, the deployment tools and the middleware implementation itself. The administrator handles the packages, the management units and the middleware as Fractal components.

6.2 "Module" systems

OSGi [21] allows the deployment of Java applications packaged in a form of *bundles* and runtime updates of those bundles. A bundle contains jar files and metadata describing those files (versioning etc.). The OSGi platform's role is to manage the lifecycle of bundles (deployment, activation, updates etc.). One of the contributions brought by the OSGi to Java community is taking jar versioning into account. The main drawback of OSGi compared to Fractal, and therefore to our solution, is that it does not provide an explicit notion of application's architecture. Moreover, OSGi services are not hierarchical and provide no control interfaces.

MJ [22] is a module system for Java. Its primary goal is to solve the issue of unexpected interactions between software components raised by large Java systems. To achieve it, MJ uses multiple class loaders but provides a high-level interface to manage these class loaders. However, MJ does not address the issue of redeployment of modules. With regards to MJ, our solution focuses mainly

on the packaging, deployment and updates of services building such large Java systems, and J2EE servers in particular. The unexpected interactions between services do not occur in JonasALaCarte since we reuse the runtime separation of services provided by the standard hierarchy of JOnAS class loaders.

7 Conclusion

In this paper we addressed the issue of service versioning and licensing in the context of J2EE application servers. We presented the use case of updating the Web container service in JOnAS J2EE server at runtime. To allow redeployment, services are packaged independently and a middleware deployment tool is used to instantiate them. Furthermore, we reconsidered the architecture of the JOnAS application server since updating a service requires that each service has a lifecycle independent from the lifecycle of the application server. We selected the Fractal component model for the redesign and the reimplementation of the JOnAS application server and its management units. This choice was driven by the fact that construction of a system with the Fractal component model yields a dynamically adaptable system where the components are units of configuration, deployment and reconfiguration.

We illustrated that by adopting a component-based approach to package the services, to build the deployment tool and to architecture the middleware itself we achieve our versioning goal in a flexible and practical way. By flexibility we understand the ability to change the deployment policies, the class loading strategies or the package storage backends by replacing the correspondent component with a new component implementing the new policy. Our solution is practical because by unifying the implementation of the packages, the deployment tool and the middleware, the management tasks from packaging to the middleware monitoring and adaptation are abstracted to the configuration of Fractal components. The manager deals with the same API for the administration of the application server in its different phases: from packaging to the dynamic adaptation at execution time. We illustrated in this paper how an administrator can navigate through the same graphical interface (Fractal Explorer) to explore a service package, to perform a redeployment operation and to check the success of the service redeployment.

Currently, we implemented the packaging and the deployment infrastructure for the middleware services. In future, we plan to apply our solution to the packaging and the deployment of the J2EE applications, as we explained in this paper. We also aim at unifying the management of the J2EE middleware and the applications. Finally, we plan to enhance the deployment process by developing new deployment policies, such as transactions, security and failure recovery, as Fractal components.

References

1. J2EE: Java 2 Platform, Enterprise Edition http://java.sun.com/j2ee/docs.html.

2. JOnAS: Java Open Application Server http://jonas.objectweb.org/.
3. Apache Tomcat http://jakarta.apache.org/tomcat/.
4. Jetty Java HTTP Servlet Server http://jetty.mortbay.org/jetty/.
5. Java Open Transaction Manager http://jotm.objectweb.org/.
6. J2EE Deployment Specification (JSR88) http://jcp.org/jsr/detail/88.jsp.
7. J2EE Management Specification (JSR77) http://jcp.org/jsr/detail/77.jsp.
8. Abdellatif, T.: Enhancing the Management of a J2EE Application Server using a Component-Based Architecture. In: Proceeding of the 31st EUROMICRO Conference (EUROMICRO'2005), Porto, Portugal (2005)
9. Bruneton, E., Coupaye, T., Stefani, J.B.: The Fractal Composition Framework (2002) http://www.objectweb.org.
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An Open Component Model and its Support in Java. In: Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004), Edinburgh, Scotland (2004)
11. The ObjectWeb Consortium http://objectweb.org.
12. The Apache Software Foundation http://apache.org.
13. Exertier, F.: J2ee deployment: The jonas case study. CoRR **cs.NI/0411054** (2004)
14. JOnAS class loader hierarchy (2004) http://jonas.objectweb.org/current/doc/.
15. Hall, R.S.: A Policy-Driven Class Loader to Support Deployment in Extensible Frameworks. In: Proceedings of the International Conference on Component Deployment (CD'2004), Edinburgh, Scotland (2004)
16. Kornas, J., Leclercq, M., Quema, V., Stefani, J.B.: Support for evolutionary changes in Java applications (2004) http://sardes.inrialpes.fr/papers/kornas04cl.pdf.
17. The Fractal Project http://fractal.objectweb.org.
18. Fleury, M., Lindfors, J.: JMX-Managing J2EE with Java Management Extensions. Sams, The JBoss Group (2002)
19. Mulder, A.: Apache Geronimo Development and Deployment. Pearson Education (2004)
20. The PicoContainer project http://www.picocontainer.org/.
21. Open Services Gateway Initiative, OSGi service gateway specification, Release 3 http://www.osgi.org.
22. Corwin, J., Bacon, D.F., Grove, D., Murthy, C.: Mj: a rational module system for java and its applications. In: Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA'2003), Anaheim, California, USA (2003)