Dynamic Adaptability of Services in Enterprise JavaBeans Architecture[*]

Zahi Jarir[*], Pierre-Charles David[**], Thomas Ledoux[**]

zahijarir@ucam.ac.ma, {pcdavid, ledoux}@emn.fr
[*] Faculté des Sciences Semlalia, Département Informatique,
BP 2390, Marrakech, Morocco
[**] École des Mines de Nantes,
4, rue Alfred Kastler, BP 20722
44307 Nantes Cedex 3-France

Abstract

To be in harmony with continuous variations of the execution environment such as fluctuating network bandwidth, decreasing battery power, and so on, new approaches are required for component-based middleware to make them adaptable with regard to these changes. The aim of the work presented in this paper is to enhance the EJB architecture by allowing applications to be aware of, and adapt to, variations in the execution context. We propose to dynamically adapt the association between EJB components and middleware services when necessary. Thus, EJB applications have the advantage to be dynamically adaptive according to changes related to their execution context.

Keywords : Adaptable Middleware, EJB architecture, Dynamic Adaptability of Services, Dynamic Reconfiguration.

1. Introduction

One of the major challenges faced by component-based middleware is the ability to dynamically adapt to many execution environment evolutions, from the change of execution platform (PDA, portable computer, etc.) to the run-time variations of availability of certain resources such as CPU, memory, communication capacities, etc.

Using a s*eparation of concerns* approach to component-based middleware development allows to distinguish functional code from non-functional code, representing middleware *services*. Then, it should possible to reconfigure the associations between these two kinds of code: that is which piece of functional code is affected (and how it is affected) by non-functional services. When these reconfigurations are guided by the evolutions of the execution environment, the resulting system can be said to be *dynamically adaptive*.

In the case of EJB environment, configuration between components and middleware services is only supported at deployment-time using a declarative deployment descriptor [1]. In this paper, we present an experiment which shows how the EJB architecture can be extended to allow reconfigurations to be performed at run-time. To introduce run-time adaptability in an EJB environment, we have reused an infrastructure for adaptable middleware developed in a previous work [2], and which relied originally on a MetaObject Protocol instead of a component model like the EJB model.

The rest of this paper is organized as follow. In Section 2, we give an overview of the existing infrastructure for adaptable middleware. In Section 3, we propose an adaptable EJB server by incorporating the main features of this infrastructure in the JOnAS EJB implementation. Finally, we conclude the paper in Section 4.

---

2. An Infrastructure for Adaptable Middleware

To ensure an advanced adaptability of component-based applications regarding changes of the environment, we have chosen to focus on the middleware layer. Because applications built on top of the middleware rely on it for all their interactions with the environment, adapting the middleware allows us to indirectly adapt applications.

To do this, we developed an infrastructure for adaptable middleware [2] (cf. Figure 1). This infrastructure considers that the application (the right side of the figure) is split in two parts: functional components (i.e. business objects) and non-functional services (corresponding to the middleware layer). The possible reconfigurations of the application consist in modifying the associations between these two parts with a very fine granularity (each component can be adapted independently). These reconfigurations are performed by a generic *adaptation engine*, which is configured for each application by *adaptation policie*s described below. Finally, the modifications are initiated by the engine when significant evolutions of the environment are detected by a simple *monitoring framework*, consisting of a collection of *probes* (CPU usage, battery life, bandwidth measure…).

Adaptation policies themselves are split in two parts:

- *System policies*, consisting in sets of rules of the form `condition ⇒ action`, where the condition is relative to the execution environment (as reified by the monitoring framework), and the action is either the attachment or detachment of a specific services, possibly with configuration parameters. These policies are relatively independent of a given application, and could be written by system administrators for example.

- *Application policies*, which defines groups of functional components according to their runtime properties (class of an object, value of a field…), and binds existing system policies to these groups. These policies must be written with a good knowledge of the semantics of the application (for example by the application programmer himself).
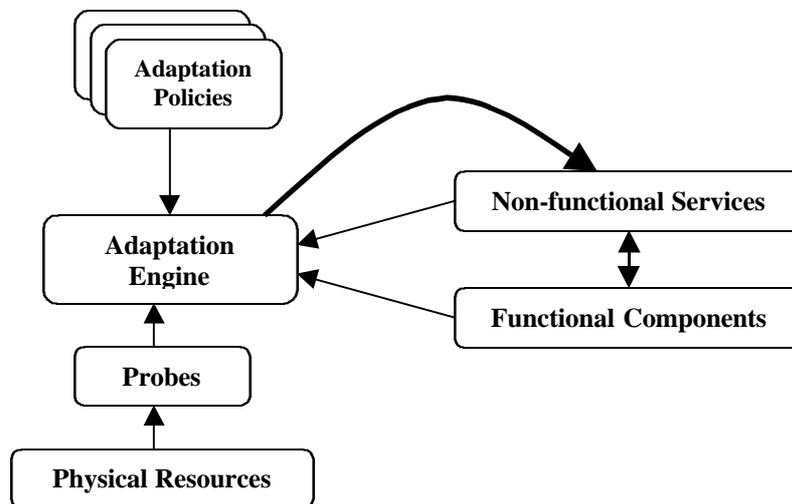

Figure 1 – An Infrastructure for middleware adaptation.

Figure 2 shows a simple example of each kind of policies, which are interpreted at run-time by the adaptation engine.

```
<system-policy name="tracer">                                   System.xml
  <rule>
   <when>
    <less-than>
      <property-value name="/system/network.bandwidth"/>
      <number value="40000"/>
    </less-than>
   </when>
   <ensure>
     <attached service="logservice" role="main"/>
   </ensure>
  </rule>
 </system-policy>
<application-policy>                                          Application.xml
  <group name="groupbean1">
    <select from="all">
       <or>
        <equals>
          <property-value name="className"/>
          <string value="sb.OpBean"/>
        </equals>
        <equals>
          <property-value name="className"/>
          <string value="sb.OpbisBean"/>
        </equals>
       </or>
    </select>
    <bind policy="tracer"/>
  </group>
</application-policy>
```

Figure 2 – Examples of a system and application policies code.

The original implementation of this infrastructure used the reflective features of a MetaObject Protocol named RAM (Reflection for Adaptable Mobility) [3], identifying base-level objects with functional components and meta-level objects with non-functional services. However, most of our infrastructure (the left side of Figure 1) is completely independent of these implementation decisions, and relies only on very generic notions of "components" and "services" which makes it reusable in other context, an EJB server for example.

Thus, thanks to the loose coupling between the different entities of our infrastructure and the particular role played by an EJB container between middleware services and EJB components (see below), we decided to replace the RAM Meta-Objects by the EJB container and reuse the monitoring framework and the decision module based on adaptation policies in order to provide dynamic adaptability of services in an EJB architecture.

3. Towards an Adaptable EJB Server

The focus of this section is on the setting up of the adaptable EJB server using the infrastructure presented in section 2. To implement this adaptable EJB server, we have used the Evidian implementation of EJB specifications named JOnAS (Java Open Application Server) [4] which is part of the ObjectWeb Open Source initiative [5]. In order to validate this adaptable EJB server, we used home-made services (for example, trace or asynchronous communication with transparent futures) rather than JOnAS predefined services, which were not implemented in a modular enough way to make them reusable for our purpose. The following subsections illustrate this implementation.

3.1. Our Approach

The EJB container is the runtime environment in which enterprise bean instances can live and execute. The enterprise bean typically contain the business logic ("functional code") for an EJB application. The associated EJB container is responsible to inject transparently services defined by the enterprise bean's deployment descriptors such as declarative transaction management, security, persistence, concurrency, and state management. Client access is mediated by the EJB container in which the enterprise bean is deployed. This layer of indirection manifests itself as an interposition object called the *EJB Object*.

Because the EJB container is an intermediary between EJB components and the outside world [1], we have decided to delegate the task of dynamic composition of services at this level. However, containers are generated statically using the information provided by the beans' deployment descriptors, which mean that we can not modify associations between beans and services at run-time without modifying these containers.

Thanks to GenIC (Generate Interposition Classes) tool offered in open source by JOnAS and allowing to generate container code, the setting up of this implementation is proved possible.

3.2. Extending the EJB Container for Adaptation

In order to respect EJB container specification, we have introduced an indirection from *EJB Object* towards another object, called *DynamicComposite*. DynamicComposite object will be responsible for playing the role of dynamic composer of EJB services.

In the standard process to develop an application for the JOnAS EJB platform, the application developer, once his beans are written, creates two XML deployment descriptors: the standard `ejb-jar.xml` common to every EJB implementation, and a JOnAS-specific `jonas-ejb-jar.xml`. These files are then read by the GenIC tool to generate the corresponding container code.

Because the indirection we want to introduce is not required for every bean and because of the negative impact it can have on performance, we wanted the developer to identify explicitly which of his beans would be dynamically adaptable. To this end, we introduced a new XML tag in the `jonas-ejb-jar.xml` file with the following syntax: `<DynamicComposite>`*true* or *false*`</DynamicComposite>`. When this tag is present and set to `true`, it means that the corresponding bean will benefit from our dynamic adaptation service.

To take this new tag into account, we had to modify the GenIC tool to generate the correct indirection on every business method call on the adaptable beans. Normally, when a EJB container intercepts a business method call it forwards the message to its managed bean (in addition to the execution of the statically required services). With our modification, the message is sent instead to a `DynamicComposite` object associated to the container. As its name implies, this object is able to compose dynamically attached or detached home-made services before or after finally sending the message to the bean.

The following example shows a (simplified) fragment of code from an EJB container (the class JonasOpRemote representing the EJB Object in JonAS) with dynamic adaptation enabled. The commented line shows the code that would have been generated if it had been disabled, or by the non-modified GenIC tool.

```
public synchronized void buy(int p1) throws RemoteException {
  sb.OpBean eb = (sb.OpBean) ctx.getInstance();
  // eb.buy(p1); // Original code
  Object[] args = new Object[]{ new Integer(p1) };
  Class[] types = new Class[]{ Integer.TYPE }
  Method meth = eb.getClass().getMethod("buy", types);
  dynamicComposite.execute(eb, meth, args);
}
```

3.3. Setting up a Dynamic Adaptation Service within the JOnAS Server

Now that we have introduced an indirection into the EJB containers, this subsection describes how we connected the `DynamicComposite` object to our existing adaptation infrastructure presented before.

Fortunately, JOnAS is written in a relatively modular way and makes it possible to add new services in addition to those defined in the EJB specification. Thus, we created a new JOnAS service, called `DynamicAdaptation`, which plays the role of a meta-service and which encapsulates our existing infrastructure (the adaptation engine and monitoring framework, see Fig. 1) as shown in Figure 3. On startup, the JOnAS server initializes all the services it is aware of, including the DynamicAdaptation one. It is during this initialization that the adaptation policies are loaded and the adaptation engine started up.

A final step is then required to close the loop by making the adaptation engine aware of the beans and their `DynamicComposite` object. First, a simple *Adapter* design pattern [6] has been developed to allow the adaptation engine to deal with `DynamicComposite` and the services it composes. Then, GenIC was again used to modify the constructors of EJB containers so that they instanciate a `DynamicComposite` and register it into the adaptation engine, through the `DynamicAdaptation` service:

```
public JOnASOpRemote(JSessionHome home) throws RemoteException {
   super(home, true);
   dynamicComposite = new DynamicComposite("sb.OpBean");
   ServiceManager sm = ServiceManager.getInstance();
   dynAdapt = (DynamicAdaptation) sm.getService("DynamicAdaptation");
   dynAdapt.registerDynamicComposite("sb.OpBean", dynamicComposite);
}
```
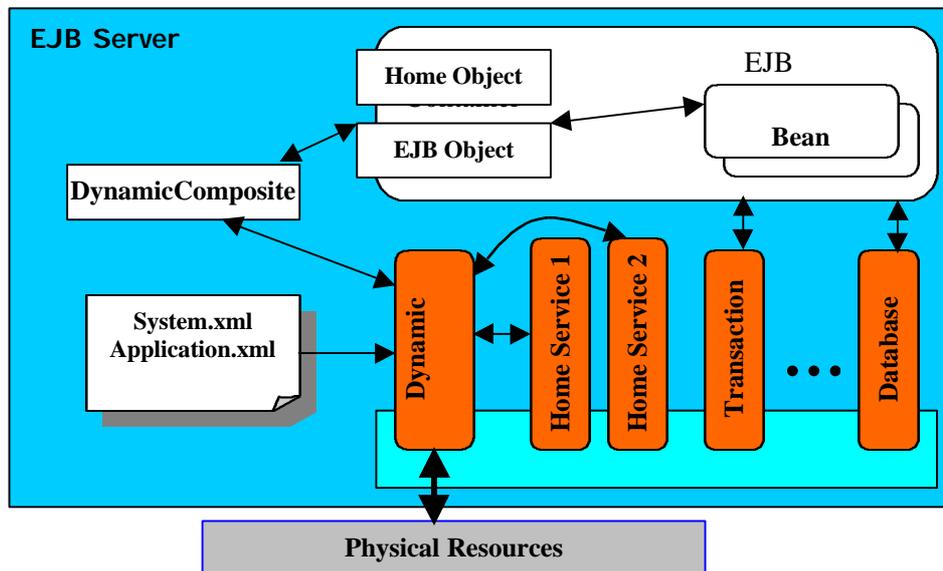

Figure 3 – An adaptable EJB architecture.

3.4. Usage

From the application developer's point of view, the resulting systems is used in much the same way as the original JOnAS. After having developed the beans normally and written the standard deployment descriptors, the programmer must decide which beans will require dynamic adaptation, and for those (and only those), add a `DynamicComposite` tag (with a `true` value to the *jonas-ejb-jar.xml* file), and then normally run the GenIC tool to generate the container classes. At this point, and even without any adaptation policy

defined, the application is ready to run in the exact same way as in the original JOnAS setup (except for the slowdown due to the unused indirection). To use the full power of our infrastructure, the programmer must write (or reuse) system and application adaptation policies that will be loaded by the DynamicAdaptation service who knows where to find them.

When the adaptation engine is started during the server initialization, it loads the provided policies, and sets up the monitoring framework to listen to the environmental conditions specified in system policies. Once this is done, the engine enters in a passive state, where it waits to be notified of environment changes by the monitoring framework (which runs asynchronously).

4. Conclusion and Perspectives

In response to the changes of the execution environment, component-based applications have to be dynamically adaptive. To attain this objective in the context of EJB environment, this paper presents an implementation of an adaptable EJB architecture based on JOnAS by integrating the main features of an existing infrastructure for adaptable middleware. This adaptable EJB architecture consists in modifying at run-time and with a fine granularity the association between EJB and middleware services. Therefore EJB applications will have the advantage to be adaptive towards any significant evolutions of the environment changes.

As mentioned previously, this architecture have been validated using home-made services. The next logical step would be to replace these home-made services by the standard EJB services. However, this is probably not possible without breaking the EJB model. First, the use of JOnAS predefined services will confront us to the complex nature of their composition. Then, the different kinds of beans provided by the EJB specification (entity, session) correspond to predefined configurations of non-functional services, which must be defined at deployment time and can not be modified. If the EJB model does not evolve towards more dynamicity, the only run-time adaptations it allow would concern services outside its scope.

5. References

[1] Linda G. DeMichiel, L. Ümit Yalçinalp¸ Sanjeev Krishnan, — Enterprise JavaBeans Specification, Version 2.0, Final Release, Sun Microsystems, August 2001.

[2] Pierre-Charles David and Thomas Ledoux — An Infrastructure for Adaptable Middleware, *submitted paper* to the Third International Workshop on Software Engineering and Middleware (SEM 2002), ICSE 2002 Co-Located Events.

[3] N. M. N Bouraqadi-Saâdani, T. Ledoux and M. Südholt — A Reflective Infrastructure for Coarse-Grained Strong Mobility and its Tool-Based Implementation, Invited presentation at the *International Workshop on ``Experiences with Reflective Systems''* (held in conjunction with Reflection 2001).

[4] JOnAS, http://www.evidian.com/jonas.

[5] ObjectWeb consortium, http://www.objectweb.org.

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides — *Design Patterns*, Addison-Wesley Reading, Massachusetts, 1995.