The JBoss Extensible Server

Marc Fleury[1] and Francisco Reverbel[2]

[1] The JBoss Group, LLC
Suite 211, 3500 Piedmont Road, NE, Atlanta, GA 30305, USA
marc@jboss.org
[2] Department of Computer Science, University of São Paulo
Rua do Matão 1010, São Paulo, SP 05508-900, Brazil
reverbel@ime.usp.br

Abstract. JBoss is an extensible, reflective, and dynamically reconfigurable Java application server. It includes a set of components that implement the J2EE specification, but its scope goes well beyond J2EE. JBoss is open-ended middleware, in the sense that users can extend middleware services by dynamically deploying new components into a running server. We believe that no other application server currently offers such a degree of extensibility. This paper focuses on two major architectural parts of JBoss: its middleware component model, based on the JMX model, and its meta-level architecture for generalized EJBs. The former requires a novel class loading model, which JBoss implements. The latter includes a powerful and flexible remote method invocation model, based on dynamic proxies, and relies on systematic usage of interceptors as aspect-oriented programming artifacts.

1 Introduction

Application servers are middleware platforms for development and deployment of component-based software. The application server offers an environment in which users can deploy *application components* — software components, developed either by the users themselves or by third-party providers, that correspond to server-side parts of distributed applications. Most application servers implement one of the industry standards currently adopted for server-side application components: Java 2 Enterprise Edition (J2EE), .NET, and the CORBA Component Model. Each of these standards defines a component model suitable for a class of application components[1].

There is no reason, however, to employ component-based techniques on user applications only. Researchers have presented compelling arguments for also exploiting these techniques *within* the middleware platform [4,17]. We claim that application servers themselves can (and should) be built in a component-based way, out of dynamically deployable components that provide middleware services to application components. On such a server, extensible and dynamically

[1] J2EE actually encompasses two application component models: Servlets/JSP, for web components used by HTTP clients, and Enterprise JavaBeans (EJB), for business components used either by RMI clients or by local clients.

reconfigurable, two general kinds of components can be deployed: middleware components and application components. In this approach, most of the "application server functionality" is actually realized by a set of middleware components deployed on a minimal server. Due to the requirement differences between middleware components and application components, multiple component models are likely to coexist in a component-based application server: a model for middleware components, plus one or more models for application components.

This paper discusses the design and implementation of JBoss, the extensible, reflective, and dynamically reconfigurable Java application server that pioneered the approach we have just outlined. The JBoss project is at the confluence of research areas such as component-based software development [28], reflective middleware [17], and aspect-oriented programming [8], all of them currently targeted by intense activity. It produced an open-source server whose download statistics [23] since January 2002 have been above 100,000 per month, and have exceeded 200,000 downloads in peak months. The JBoss server includes a set of middleware components that implement the J2EE specification [26], but its scope goes well beyond J2EE. JBoss is open-ended middleware, in the sense that users can extend middleware services by dynamically deploying new components into a running server. To the best of our knowledge, no other application server offers this degree of extensibility.

1.1 The Foundation

An emerging standard, the Java Management Extensions (JMX) [27] specification, provides the foundation for JBoss middleware components. JMX defines an architecture for dynamic management of resources (applications, systems, or network devices) distributed across a network.

In JMX, as in other management architectures, resources must be instrumented to become manageable. One instruments a resource by associating one or more management components with the resource. *Dynamic* management [20] means that one must be able to dynamically load, unload, and evolve these components, without stopping the applications, systems, or devices they instrument. These key requirements for a dynamic management architecture have similar counterparts in the more general field of adaptive middleware.

JMX was chosen as the basis of the JBoss component model for the following reasons: (*i*) it provides a lightweight environment in which components — as well as their class definitions — can be dynamically loaded and updated, (*ii*) it supports component introspection and component adaptation, (*iii*) it decouples components from their clients, allowing components to adapt, and their interfaces to evolve, while their clients are active, (*iv*) it can be used as a realization of the microkernel architectural pattern [2], to provide a minimal kernel that serves as a software bus for extensions, possibly developed by independent parties, and (*v*) its usage makes JBoss manageable through JMX-compliant applications.

1.2 The Building

On top of JMX, JBoss introduces its own model for middleware components, centered on the concept of *service component*. The JBoss service component model extends and refines the JMX model to address some issues beyond the scope of JMX: service lifecycle, dependencies between services, deployment and redeployment of services, dynamic configuration and reconfiguration of services, and component packaging.

Nearly all the "application server functionality" of JBoss is modularly provided by service components plugged into a JMX-based "server spine". Service components implement every key feature of J2EE: naming service, transaction management, security service, servlet/JSP support, EJB support, asynchronous messaging, database connection pooling, and IIOP support. They also implement important features not specified by J2EE, like clustering and fail-over.

Rather than attempting to present each of these subsystems, this paper focuses on the EJB subsystem, which we consider a particularly interesting use case for service components. JBoss supports a generalization of the EJB model by using service components as meta components. Its meta-level architecture for generalized EJBs is built upon four kinds of elements: invokers, containers, dynamic proxies, and interceptors. *Invokers* are service components that provide a general remote method invocation service over a variety of protocols. *Containers* are service components that enhance application component classes with predefined and packaged sets of aspect requirements. They provide server-side join points for aspects [16,8] that crosscut the central concerns of multiple EJB components. *Dynamic proxies*, used as client stubs, provide similar join points at the client side. *Interceptors* implement crosscutting aspects at both sides. Containers, proxies, and interceptors are neither created nor manipulated by initiatives of the server spine, but by actions of an *EJB deployer*, which is a service component itself. In other words, EJB support is pluggable.

1.3 Organization of This Paper

The next section reviews the elements of the JMX architecture that are essential to the understanding of the JBoss service component model. Section 3 presents service components; Sect. 4 describes the meta-level architecture through which JBoss supports generalized EJB components; Sect. 5 summarizes the history of the JBoss project; Sect. 6 discusses ongoing and future work; Sect. 7 examines related work; and Sect. 8 presents our concluding remarks.

2 JMX Foundation

The JMX architecture is shown in Fig. 1. It consists of three levels: the instrumentation level, the agent level, and the distributed services level.

- The *instrumentation level* defines how to instrument resources so that they can be monitored and manipulated by (possibly remote) management applications. The instrumentation of a given resource is provided by one or more *managed beans* (*MBeans*), Java objects that conform to certain conventions and expose a *management interface* to their clients.

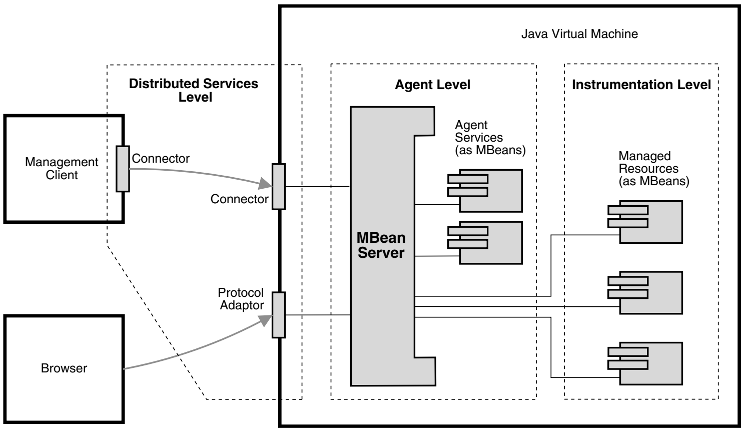

Fig. 1. The JMX architecture

- The *agent level* defines an agent that manages the set of instrumented resources within a Java virtual machine, in behalf of (possibly remote) management applications. The *JMX agent* consists of an in-process server, the *MBean server*, plus a standardized set of agent services: dynamic class loading, monitoring, timer service, and relation service. Agent services are implemented as MBeans; this makes them manageable through the MBean server, like user resources.
- The *distributed services level* specifies how management applications interact with remote JMX agents and how agent-to-agent communication takes place. It consists of *connectors* and *protocol adaptors*, implemented as MBeans. This level is not fully defined at the present phase of the JMX specification process. For information on connectors and protocol adaptors, see [19].

Together, the instrumentation and agent levels define an *in-process* component model. The MBean server provides a registry for JMX components (MBeans) and mediates any accesses to their management interfaces. At registration time, each MBean is assigned an *object name* that must be unique in the context of the MBean server. In-process clients use object names (rather than Java references) to refer to MBeans. To invoke a management operation[2] on an MBean, a local client (typically another MBean) uses the object name of the target MBean. It passes the object name as the first argument in a call to the MBean server's `invoke` method, whose declaration[3] is

```
Object invoke(ObjectName targetName,
              String operationName,
              Object[] params,
              String[] signature);
```

[2] *Management operations* are operations that belong to the MBean's management interface.

[3] For clarity, we have omitted any `throws` clauses from the method declarations presented in this paper.

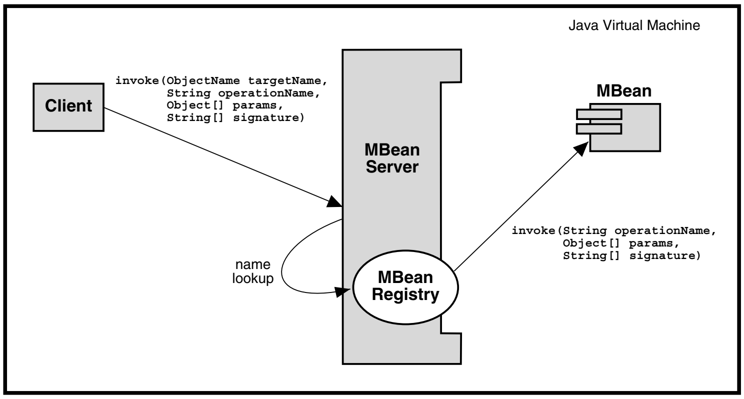
Java Virtual Machine
Client
invoke(ObjectName targetName, String operationName, Object[] params, String[] signature)
MBean Server
name lookup
MBean Registry
invoke(String operationName, Object[] params, String[] signature)
MBean

Fig. 2. Method invocation on a dynamic MBean

The MBean server looks up the object name in its registry and forwards the invocation to the target MBean. Figure 2 illustrates this process for a target MBean with a method quite similar to the MBean server's `invoke` method. As we will see shortly, this is the case of a dynamic MBean.

The MBean server introduces a level of indirection that decouples MBeans from their clients. In-process clients of an MBean do not need direct Java references to it. Moreover, clients need no information on the MBean's Java class, nor do they need information (not even at runtime) on the Java interfaces the MBean implements. All they need is the MBean's object name, plus knowledge (possibly obtained at runtime) of its management interface. (We have not yet discussed what would be the management interface of an MBean, but the preceding assertions imply that it does not necessarily correspond to a Java interface.) This very simple arrangement favors adaptation: the absence of references to an MBean scattered across its clients facilitates the replacement of that MBean; the absence of client knowledge about its class and its Java interfaces enables dynamic changes both to the implementation and to the management interface of the MBean.

The *management interface* of an MBean consists of four parts: (*i*) management attributes, whose values are accessible to clients through the MBean server; (*ii*) management operations, which clients can invoke through the MBean server; (*iii*) notifications emitted by the MBean and delivered to registered listeners; (*iv*) constructors, defined by the MBean's Java class.

2.1 Dynamic MBeans and Standard MBeans

JMX supports two kinds of MBeans,[4] which differ on how they expose to the MBean server their management attributes and their management operations.

[4] The JMX specification uses the expression "types of MBeans". We prefer "kinds of MBeans", to avoid confusion with Java types.

```
interface DynamicMBean {
    Object getAttribute(String attrName);
    AttributeList getAttributes(String[] attrNames);
    void setAttribute(Attribute attr);
    AttributeList setAttributes(AttributeList attrs);
    Object invoke(String opName, Object[] params, String[] signature);
    MBeanInfo getMBeanInfo();
}
```

Fig. 3. The `DynamicMBean` interface

One kind of MBean (the so-called *dynamic* kind) implements a predefined Java interface, regardless of its management interface, and relies on metadata to specify its management interface. The other kind (the so-called *standard* kind) implements a Java interface defined after — and determined by — the MBean's management interface. The kind of an MBean is an implementation detail hidden from clients, which access both kinds of MBeans exactly in the same way.

Dynamic MBeans. By implementing the interface shown in Fig. 3, a *dynamic MBean* provides generic methods for attribute access and operation invocation.

The method `getMBeanInfo()` returns a self-description of the MBean. The metadata class `MBeanInfo` supports *MBean introspection*, i.e., from an `MBeanInfo` instance one can obtain complete information about the management interface of the MBean described by that instance: attribute names and types, operation names and signatures, notification types, Java class name, and Java constructor signatures. The management attributes and operations supported by a dynamic MBean do not necessarily correspond to Java fields and methods, nor do they need to be associated with methods of some particular Java interface implemented by that MBean. They are specified solely by the dynamic MBean's self-description.

Standard MBeans. A *standard MBean* exposes its management attributes and operations by implementing a Java interface named after the MBean's Java class, with the suffix `MBean`. This interface follows JavaBean-like rules to represent those attributes and operations: it must have a get method for each readable attribute, a set method for each writable attribute, and an additional method for each operation.

As an example, a standard MBean of class `Foo` must implement a Java interface named `FooMBean`, whose definition is determined by the MBean's management interface. Suppose the management interface has one attribute and one operation: an integer-valued read/write attribute, whose name is `"Count"`, and an operation named `"doSomething"`, which receives a `long` parameter and returns a `double` value. Then `FooMBean` must be the following Java interface:

```
interface MBeanServer {
    ObjectInstance registerMBean(Object object, ObjectName name);
    void unregisterMBean(ObjectName name);
    ...
    Object getAttribute(ObjectName name, String attrName);
    AttributeList getAttributes(ObjectName name, String[] attrNames);
    void setAttribute(ObjectName name, Attribute attr);
    AttributeList setAttributes(ObjectName name, AttributeList attrs);
    Object invoke(ObjectName name, String opName,
                  Object[] params, String[] signature);
    MBeanInfo getMBeanInfo(ObjectName name);
}
```

Fig. 4. The `MBeanServer` interface

```
interface FooMBean {
    int getCount();
    void setCount(int value);
    double doSomething(long param);
}
```

Why Two Kinds of MBeans? Standard MBeans are easier to implement, because they relieve their writers from the task of building metadata instances to describe management interfaces. On the other hand, dynamic MBeans are more flexible, as the definitions of their management interfaces can be postponed until runtime. Both kinds of MBeans support some form of evolution of management interfaces. In the case of a standard MBean, however, evolution requires object replacement: one must bind to the MBean's object name an instance of another Java class. Dynamic MBeans support evolution *without* object replacement.

2.2 The MBean Server

The `MBeanServer` interface appears in Fig. 4. The first two methods shown are a manifestation of the MBean server's role as a registry for MBeans. The remaining ones form a group that parallels the interface `DynamicMBean`; each of them receives as its first parameter an `ObjectName` that specifies the target MBean. For brevity, we have omitted several methods, including the ones for MBean creation and those related with JMX notifications.

Clients access MBeans through the MBean server, using its `DynamicMBean`-like methods. The server's `getMBeanInfo` method returns metadata that describes the management interface of the target MBean, regardless of kind. Its return value is either a dynamic MBean's self-description or an `MBeanInfo` instance constructed by the server, most likely at MBean registration time. In the latter case, the `MBeanInfo` instance contains information obtained through

Java introspection on the `ClassNameMBean` interface implemented by a standard MBean whose class is `ClassName`.

The methods for attribute access and operation invocation act differently depending on the kind of the target MBean. If the target is a dynamic MBean, the MBean server simply forwards the invocation to the target MBean through the corresponding method of the `DynamicMBean` interface. If the target is a standard MBean, the MBean server converts the invocation into a suitable call to the target's `ClassNameMBean` interface. As an example, the invocation

```
mbeanServer.getAttribute(targetName, "Count");
```

would be converted into the invocation

```
targetMBean.getCount();
```

Different MBean server implementations may employ different approaches to do such conversions. The simplest approach uses the Java reflection API; in this case the `getCount()` invocation above would be performed as a reflective call to the `ClassNameMBean` interface. The MBean server included in JBoss takes another approach: it avoids the extra cost of reflective calls by applying byte code generation techniques [5]. At MBean registration time, the MBean server performs Java introspection on the `ClassNameMBean` interface implemented by a standard MBean and generates the class of a suitable object adapter [11]. The generated adapter implements the `DynamicMBean` interface and issues non-reflective calls to methods of the `ClassNameMBean` interface.

2.3 Reflection in JMX

JMX can be regarded as a reflective architecture. The method `getMBeanInfo` supports MBean introspection. Object replacement is a simple yet effective form of adaptation at the MBean level: clients refer to MBeans by object names, so the perceived behavior of any MBean can be changed by object replacement. Other forms of adaptation are possible for dynamic MBeans. The agent level includes a dynamic class loading service that facilitates object replacement; this service allows MBeans to be instantiated using new Java classes loaded from remote servers.

3 Service Components

JMX does not include mechanisms for managing dependencies between MBeans, nor does it define a concept of service lifecycle for MBeans. Packaging and deployment of components are also out of the scope of the JMX specification. JBoss addresses all these issues with the notions of service MBean and deployable MBean. *Service MBeans* (also called *service components*) are MBeans whose management interfaces include service lifecycle operations. *Deployable MBeans* (also called *deployable services*), a JBoss-specific extension to JMX, are service

MBeans packaged according to EJB-like conventions, in deployment units called *service archives* (SARs). A service archive contains class files for one or more deployable services, plus a *service descriptor*, which conveys information needed at deployment time.

3.1 Service Lifecycle

A service component may be in the *stopped state* or in the *active state*. At each state transition, one of the following lifecycle operations is invoked on the service MBean:

- `create` — invoked once on each service MBean, after the receiver was created and registered with the MBean server. This operation tells the receiver to complete its initialization and places it in the stopped state.
- `start` — takes a service MBean from the stopped state to the active state. A `start` invocation tells the receiver to do whatever it needs to become fully operational.
- `stop` — takes a service MBean from the active state to the stopped state. A `stop` invocation tells the receiver to undo any actions it took within the `start` operation.
- `destroy` — tells the receiver to clean up its resources. This operation is invoked once on each service MBean, when the receiver is in the stopped state and is about to be unregistered from the MBean server.

The lifecycle operations supported by a service component should be exposed in its management interface. A service MBean is not required to support all four lifecycle operations. For instance, a component with no resources that require clean up does not need a `destroy` operation in its management interface. The set of lifecycle operations in a component's management interface indicates which service lifecycle events are relevant to the component.

3.2 Service Descriptors

Deployable MBeans are service MBeans packaged together with deployment information. Every service archive includes a *service descriptor*, an XML file that contains an `mbean` element for each service component in that deployment unit. An `mbean` element specifies the following information: (*i*) Java class and object name of a deployable MBean, (*ii*) initial values for (some) management attributes of the MBean, and (*iii*) dependencies from the MBean to other deployable MBeans.

Figure 5 shows a service descriptor for a deployment unit with five deployable services. The XML attributes `code` and `name` are mandatory within `mbean` elements; they specify the MBean's class and its object name[5]. The nested `attribute` elements are optional; each such element defines the initial value

[5] Strings such as `"jboss:service=WebService"` and `"jboss:service=XidFactory"` contain textual representations of JMX `ObjectNames`.

```
<server>

  <!-- Web server for class loading -->
  <mbean code="org.jboss.web.WebService"
         name="jboss:service=WebService">
    <attribute name="Port">8083</attribute>
    <attribute name="DownloadServerClasses">true</attribute>
  </mbean>

  <!-- XID factory -->
  <mbean code="org.jboss.tm.XidFactory"
         name="jboss:service=XidFactory">
    <attribute name="Pad">true</attribute>
  </mbean>

  <!-- Transaction manager -->
  <mbean code="org.jboss.tm.TransactionManagerService"
         name="jboss:service=TransactionManager">
    <attribute name="TransactionTimeout">300</attribute>
    <depends optional-attribute-name="XidFactory">
            jboss:service=XidFactory</depends>
  </mbean>

  <!-- EJB deployer -->
  <mbean code="org.jboss.ejb.EJBDeployer"
         name="jboss.ejb:service=EJBDeployer">
    <attribute name="VerifyDeployments">true</attribute>
    <attribute name="ValidateDTDs">false</attribute>
    <attribute name="VerifierVerbose">true</attribute>
    <depends>jboss:service=TransactionManager</depends>
    <depends>jboss:service=WebService</depends>
  </mbean>

  <!-- RMI/JRMP invoker -->
  <mbean code="org.jboss.invocation.jrmp.server.JRMPInvoker"
         name="jboss:service=invoker,type=jrmp">
    <attribute name="RMIObjectPort">4444</attribute>
    <depends>jboss:service=TransactionManager</depends>
  </mbean>

</server>
```

Fig. 5. Example of service descriptor file

for a writable attribute in the management interface of the MBean. The nested `depends` elements, also optional, specify dependencies from the enclosing MBean to the MBeans whose object names appear within those elements. The service descriptor in Fig. 5 specifies dependencies from the transaction manager to the XID factory, from the EJB deployer to the web server, from the EJB deployer to the transaction manager, and from the RMI/JRMP invoker to the transaction manager.

The optional XML attribute `optional-attribute-name`, which appears in one of the `depends` elements in Fig. 5, specifies the name of a writable attribute in the management interface of the enclosing MBean. This writable attribute will be set to the object name enclosed by the `depends` element. In other words, the element

```
<depends optional-attribute-name="SomeAttrName">SomeObjName</depends>
```

is equivalent to the elements

```
<depends>SomeObjName</depends>
<attribute name="SomeAttrName">SomeObjName</attribute>
```

3.3 Dependency Management

JBoss employs a variant of the component configurator pattern [22] to control the lifecycle of deployable services. Deployments of SAR files with service MBeans are handled by a `SARDeployer`, which also acts as a component configurator. A `ServiceController` plays the role of component repository, keeps track of the dependencies between deployable MBeans, and ensures that components with unsatisfied dependencies are disallowed to enter (or to remain in) the active state. The `SARDeployer` and the `ServiceController` collaborate to invoke lifecycle operations on deployable MBeans. They enforce the following protocol:

- When the `create` operation is invoked on a deployable MBean, all deployable MBeans on which the receiver depends have also had their `create` operations invoked. Moreover, the receiver's management attributes have already had their values set to the ones specified in the service descriptor. At this point the target MBean can check if required resources exist. It cannot yet use other deployable MBeans, which are not guaranteed to be operational until they have received `start` invocations.
- When the `start` operation is invoked on a deployable MBean, all deployable MBeans on which the receiver depends have also had their `start` operations invoked.
- When the `stop` operation is invoked on a deployable MBean, all deployable MBeans that depend on the receiver have also had their `stop` operations invoked.
- When the `destroy` operation is invoked on a deployable MBean, all deployable MBeans that depend on the receiver have also had their `destroy` operations invoked.

Deployment/undeployment events drive the lifecycle of deployable services. In response to these events, the `ServiceController` invokes (through the MBean server) service lifecycle operations on deployable MBeans. It issues `create` invocations to ensure that all services on which a given service depends on are created before the service is created, it issues `destroy` invocations to ensure that all services that depend on a given service are destroyed before the service is destroyed, etc.

A word on the intended usage of lifecycle operations on deployable MBeans: `start` and `stop` are expected to be lighter than `create` and `destroy`. One should think of `create` as the (re)configuration hook invoked when a component is (re)deployed, of `destroy` as the clean up hook called when a component is undeployed, and of `stop`/`start` as suspend/resume operations that are also performed at the very beginning of every undeployment (`stop`) or at the very end of every deployment (`start`).

3.4 Deployment and Undeployment

Deployable MBeans are not the only kind of deployable component that JBoss supports. Besides SARs with deployable MBeans, other types of deployment units are supported as well: plain JAR files with Java classes to be loaded into the server, resource archives (RARs) with resource adapter components, EJB-JARs with EJB components, web application archives (WARs) with servlet/JSP components, and enterprise application archives (EARs) with multi-tier applications.

A `MainDeployer` handles all deployable units by delegating the actual deployment tasks to sub-deployers specific to the various kinds of components: `SARDeployer`, `JARDeployer`, `EJBDeployer`, etc. The set of sub-deployers (and hence the set of deployment units supported) is open-ended. Sub-deployers are service MBeans that register themselves with the `MainDeployer`, which is also a service MBean. The two most fundamental sub-deployers are the `JARDeployer` and the `SARDeployer`. The `MainDeployer`, the `JARDeployer` and the `SARDeployer` are not deployable components: they are created and activated directly by the JBoss boot method. All other sub-deployers are deployable MBeans. This means, for instance, that EJB support (the ability of deploying EJB components into the application server) and servlet/JSP support (the ability of deploying web components) are dynamically deployable features themselves.

The `MainDeployer`'s management interface corresponds to the Java interface partly shown in Fig. 6. The first three operations deal with sub-deployers: registration, unregistration, and listing. The following four operations deal with deployable units of any kind: deployment and undeployment of a unit specified by an URL, check for the presence of a given unit deployed into the server, and listing of all units deployed into the server. Any of the `MainDeployer`'s operations can be invoked from management clients such as the JBoss management console, which is accessible through a web browser, or from remote management applications that access the MBean server through connectors. Moreover, the URL parameter expected by some operations may refer to a deployable unit located at some remote host. If a remote URL is passed to `deploy` operation, a remote

```
interface MainDeployerMBean extends ServiceMBean {
    void addDeployer(SubDeployer deployer);
    void removeDeployer(SubDeployer deployer);
    Collection listDeployers();
    void deploy(URL url);
    void undeploy(URL url);
    boolean isDeployed(URL url);
    Collection listDeployed();
    ...
}
```

Fig. 6. Management interface of the `MainDeployer`

unit will be downloaded to the application server host and then dynamically deployed into JBoss.

Hot Deployment. Deployable components can be conveniently deployed into JBoss simply by dropping deployment units into a well-known directory. This feature is called *hot deployment*. A `DeploymentScanner` monitors the files in that directory and handles every deployment unit found. Not surprisingly, the `DeploymentScanner` is a deployable MBean itself. A thread started by this MBean repeatedly scans the deployment directory and invokes the `MainDeployer` whenever it detects a change. The addition of a new file causes a `deploy` invocation, the removal of an existing file causes an `undeploy` invocation, and a change to an existing file change causes a redeploy (`undeploy` followed by `deploy`) operation.

Class Visibility. We say that a class is *visible* within a server either if the class is already loaded in the server's virtual machine or if it is loadable through the class loading scheme used by the server. For instance, a class in the system classpath might not be loaded, but it is certainly visible.

Unlike servers in which class visibility is statically provided by the system class loader, used to load classes once and for all, JBoss allows visible classes to vary over time. Hot deployment establishes a causal connection from the presence of deployment units in the deployment directory to the visibility, within the server, of the classes contained in those deployment units[6]. If the contents of the deployment directory change, then the set of classes that are visible within the server will also change. Modifications to the deployment directory can cause a given (fully qualified) class name to be associated with different class types at different points in time.

Recall that a Java class type is uniquely determined by the combination of a class loader and a fully qualified class name [18]. In response to changes in the

[6] The deployment directory is not the only factor that determines class visibility, which is affected by direct calls to the `MainDeployer` as well.

deployment directory, JBoss instantiates class loaders to dynamically define new class types, which may correspond to “new versions” of previously defined class types.

3.5 Class Loading Issues

A number of Java application servers support some form of dynamic deployment for *application* components. They use variants of a class loading approach that could be called *loader-per-deployment* [21,14]. A separate class loader, constructed at deployment time and bound to a deployment unit, is assigned to each deployment[7]. This class loader, usually a `java.net.URLClassLoader`, loads class files from the deployment unit it is bound to.

The loader-per-deployment scheme creates a separate namespace for each set of classes loaded from some deployment unit. Components loaded from different deployment units may contain classes with the same name, but these classes will be treated as different types by the Java virtual machine [18]. This approach avoids class clashes between deployment units, but hinders interactions between separately deployed parts. Even though separate namespaces might be convenient for application components, we argue that they are ill-suited for the dynamically deployable parts of an extensible system such as JBoss.

Problems with Hierarchical Loader-per-Deployment Approaches. In order to interact within a Java virtual machine, components need to share non-system classes. The class loader parent delegation model[8] implies that any set of interacting components must be loaded by a set of class loaders with a common ancestor, which loads the collection of classes shared among all those components. Components that share non-system classes require a hierarchical deployment process, in which shared class files are somehow deployed before the components that will share them. Such a process would correspond to a “deployment tree” rooted at the set of shared class files. This approach, however, fosters replication of class files across deployment trees associated with independent (non-interacting) sets of components. Most importantly, experience with earlier versions of JBoss has shown us that hierarchical loader-per-deployment schemes are cumbersome in dynamic environments, specially in the presence of interactions between middleware components developed by different teams.

If a set of components needs to share some non-system class C, then there are dependencies from the component that provides class C to all other components in the set. Component dependencies form a directed acyclic graph, which in general cannot be reduced to a deployment tree. Moreover, dependencies may

[7] This description applies to “simple” deployment units (EJB-JARs or WARs). In the case of a “composite” deployment unit (EAR) with multiple EJB and web modules, more than one class loader will be actually created. See [14] for details.

[8] In this model [12], every class loader has a parent class loader. Whenever a class loader is asked to load a class, it first delegates the request to its parent. If the parent fails to load the class, then the class loader attempts to perform the task itself.

change over time, as components are updated. So even if a given set of components currently has a rooted tree as its dependency graph, nothing ensures that future updates to the components will not break the hierarchic structure of that graph. A non-hierarchical approach to class loading is needed to accommodate the general nature of component dependencies.

The J2EE Solution. In spite of their shortcomings, loader-per-deployment schemes are used for application components, by application servers that forbid changes to the individual parts of an application. A component-based application (a complete set of interacting parts) must be deployed as a whole into such a system, so that its components can be loaded by the same class loader or by a suitable class loader hierarchy. Packing together parts that work together is still considered an acceptable rule for application scenarios; it is actually the standard J2EE practice.

3.6 Unified Class Loaders

JBoss employs a new class loader architecture that facilitates sharing of classes across deployment units. A collection of *unified class loaders* acts as a single class loader, which places into a single namespace all classes it loads. Rather than creating its own namespace, each unified class loader adds `Class` objects to a flat namespace shared among all unified class loaders. This is a significant departure from the hierarchical class loading model introduced in JDK 1.2[9].

Instances of `UnifiedClassLoader`, a subclass of `java.net.URLClassLoader`, are registered with a `UnifiedLoaderRepository` MBean. This collection of class loaders behaves like a special kind of `java.net.URLClassLoader` that allows its array of URLs to be updated at any time[10]. To add an URL, create a new `UnifiedClassLoader` for the URL, and register the class loader with the repository. To remove an URL, remove the corresponding `UnifiedClassLoader` from the repository. To load classes, use any of the `UnifiedClassLoaders` in the repository. They are all equivalent and share a single namespace.

Conceptual Description. The class loading strategy is conceptually very simple. The unified loader repository maintains loaded classes in a cache implemented by two hash maps: one that maps class names into `Class` objects, and another that maps class loaders into sets of class names. When a unified class loader is asked to load a class, it first looks at the repository it is registered with and checks if the class is already in the repository's cache. If the class is not cached, the unified class loader attempts to load a class file from its URL. In

[9] JBoss adds `Class` objects to a flat namespace by default, but is also supports "scoped class loading", which creates new namespaces, to allow for concurrent versioning of EAR deployment units. By explicitly specifying scoped class loading, users can have different versions of the same components running simultaneously into a server.

[10] A `java.net.URLClassLoader` has a constant array of URLs, specified by a constructor parameter, and loads classes from these URLs.

case it does not find the class file, it iterates through the class loaders registered with the repository until one of them loads the class file. The repository updates its cache whenever a class file is loaded by some unified class loader. On the removal of a class loader from the repository, all classes loaded by the class loader should be removed from the repository's cache. The map from class loaders to sets of class names serves this purpose.

Locking Issues. The actual implementation of unified class loaders, however, is complicated by locking issues. In an attempt to ensure that concurrent threads never load the same class more than once, Java virtual machines typically lock a class loader while the loader is loading a class[11]. Only one thread at a time is allowed to load a class using a given class loader. Under such locking policy, deadlock would occur if the conceptual class loading strategy just described were literally translated into a naive implementation. Suppose that a thread t_1 uses some unified class loader l_1 to load a class c_1 and, at the same time, a thread t_2 uses some unified class loader l_2 to load a class c_2. Assume that neither c_1 nor c_2 is in the repository's cache. Moreover, suppose that the class file for c_1 is in l_2's URL and the class file for c_2 is in l_1's URL. A naive implementation of unified class loaders would deadlock in this scenario.

Deadlock Avoidance. The JBoss implementation avoids deadlocks by using a task scheduler that allocates class loading tasks to a pool of cooperating threads whose elements are temporary owners of unified class loaders. A thread is in the pool as long as it holds some unified class loader's monitor lock. By repeatedly calling the task scheduler to get its next task, each such thread sequentially processes all class loading tasks that must be handled by its unified class loader, including the ones initiated by other threads[12].

[11] Depending on the class loaders involved, locking class loaders might actually be a futile attempt to prevent concurrent threads from loading a class more than once. In the case of unified class loaders, locks on class loaders are superfluous and not effective, for the repository is what needs to be locked to ensure that no class gets loaded twice. There should be a way to tell the Java virtual machine not to lock these class loaders. This would be possible if the class loading method directly called by Sun's virtual machines — the method `loadClassInternal` — were not defined as private and synchronized in class `java.lang.ClassLoader`. The method `loadClassInternal` should be protected, rather than private, so that specialized subclasses of `ClassLoader` (such as `UnifiedClassLoader`) could redefine it with no synchronization. With respect to this issue, a member of the JBoss team has filed with Sun a bug report [15] against the Java runtime environment.

[12] If there were a way to tell the Java virtual machine not to lock unified class loaders, then there would be no need for such a deadlock avoidance scheme. Explicit task scheduling appears here merely as an elaborate trick through which JBoss circumvents the Java bug mentioned in the preceding footnote.

3.7 Dynamic Proxy Usage

Since JDK 1.3, the Java reflection API supports a limited form of program adaptation: the late definition of certain object adapter classes called *dynamic proxy* classes. A program can dynamically define a proxy class that implements given interfaces by delegating all method invocations to a generic *invocation handler*, through a type-independent interface. A dynamic proxy instance can be regarded as an object adapter [11] that converts the type-independent interface of its invocation handler into a list of interfaces specified at runtime.

Close interplay takes place between application components, whose interfaces are not known until runtime, and the middleware components that make up the JBoss server. Most of the time such collaboration happens transparently with respect to application components, which perceive themselves as interacting with each other and only occasionally take explicit actions to request middleware services. Enabling transparent collaboration between application components and middleware components, within a platform written in a strongly-typed language like Java, poses a problem: the platform must somehow bridge the gap between the interfaces that are application-specific and those exposed by middleware components. Many application servers bridge this gap with classes statically generated through compilation-based approaches, at the expense of flexibility and developer friendliness. Nevertheless, this is not an option for a system intended to support dynamic deployment of application components whose interfaces are not known in advance. Dynamic proxies are crucially important for JBoss because they can bridge that gap at runtime.

The Dynamic Stub Idiom. Recall how Java RMI handles parameter and return value passing: serializable types are normally passed by value, remote types are normally passed by reference. A *serializable type* is either a primitive type or an instance of a class that implements `java.io.Serializable` or `java.io.Externalizable`. Serialized types are normally passed to other virtual machines in serialized form. A *remote type* is an instance of a class that implements `java.rmi.Remote`. When an RMI parameter or return value is a remote type, a stub for the remote object is normally passed instead of the object.

Note the occurrences of the adverb "normally" in the preceding paragraph. What happens in the case of a remote object that is also serializable? If the object has not been exported (made available to remote clients) through the RMI system , then it will be passed by value. Passing remote objects by value, in serialized form, allows the creation of *custom stubs*. Rather than using stubs generated by RMI tools, a programmer can create his own stub objects, which interact over a *custom protocol* with the remote objects that they represent. Custom stubs should be remote (but not exported) and serializable, so they are passed by value to other virtual machines.

Besides interacting with the remote objects they represent, custom stubs must also implement the application-specific interfaces of these objects. Rather than writing (or creating a tool that generates) a custom stub class for every application interface, one can use dynamic proxies as custom stubs. This is the *dynamic stub idiom.* A dynamic proxy implements the interfaces expected by the application. The invocation handler is the "customized" part of the stub: it uses a custom protocol to interact with the remote object represented by the stub. Dynamic stubs are typically created at the server side and passed by value to other virtual machines. This is possible because dynamic proxy instances are serializable, provided that their invocation handlers are serializable. Section 4.2 describes the key role of dynamic stubs in JBoss.

Conversion of Management Interfaces into Java Interfaces. This is a more prosaic (and very common) usage of dynamic proxies in JBoss. In this use case, a dynamic proxy instance, associated with a given MBean, implements a Java interface whose methods correspond to (possibly a subset of) the MBean's management interface. The proxy has an invocation handler that knows the MBean's object name and forwards method invocations to the MBean, through the MBean server.

4 Meta-level Architecture for Generalized EJBs

The conceptual definition of the EJB architecture [25] relies strongly on the abstract notion of an *EJB container*. In JBoss, a set of meta-level components works together to implement this conceptual abstraction. A *generalized EJB container* is a set of pluggable aspects that can be selected and changed by users. Extended EJB functionality is supported by a meta-level architecture whose central features are:

- usage of MBeans as meta-level components that support and manage base-level EJB[13] components;
- a uniform model for reifying base-level method invocations;
- a remote method invocation model based on dynamic proxies;
- usage of a variant of the interceptor pattern [22] as an aspect-oriented programming [16,8] technique.

In what follows, we will adopt an EJB standpoint and consider that the base level consists of EJB components. Accordingly, we will refer to EJB interfaces as base-level interfaces. From this perspective, MBeans belong to the meta level, and their management interfaces are meta-level interfaces. Figure 7 shows the meta-level architecture whose elements will be discussed in the following subsections.

4.1 Reified Method Invocations

Interactions between base-level components follow a variant of the message reification model [9]. Inter-component method invocations performed at the base

[13] For brevity, we refer to our base-level components simply as EJBs, but it will shortly become clear that they are actually generalized EJBs.

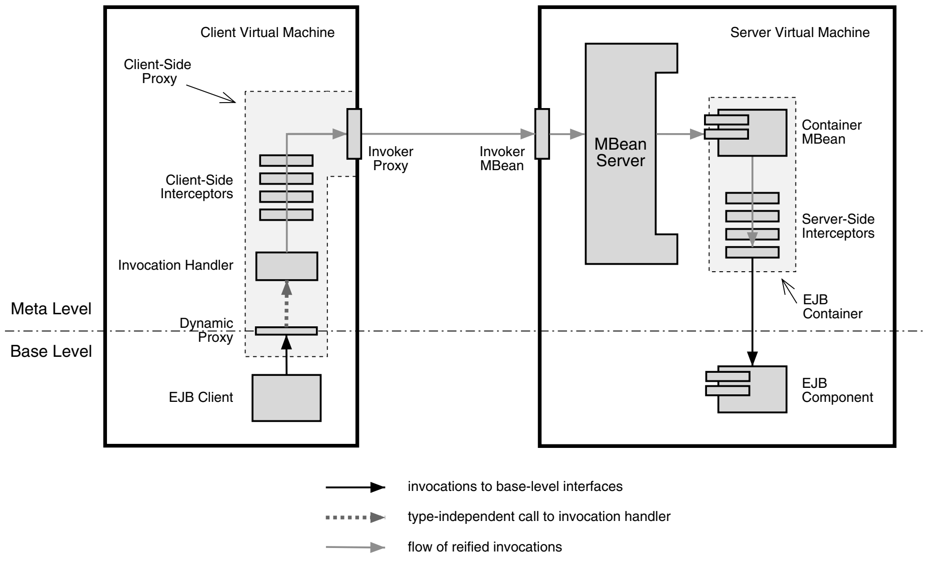

Fig. 7. Meta-level architecture for EJB

level are reified by special `Invocation` objects. Dynamic proxies receive all EJB invocations executed by EJB clients (which may be EJBs themselves) and shift those invocations up to the meta level, by transparently converting them into `Invocation` objects[14].

The gray arrows in Fig. 7 show the flow of reified invocations. The invocation handler creates a reified invocation whenever a method call is issued on the client-side proxy. After traversing a chain of *client-side interceptors*, each reified invocation is sent by an *invoker proxy* to an *invoker MBean* at the server-side, where it is routed through the *container MBean* associated with the target EJB.

Figure 8 lists the fields of a reified invocation. The `objectName` field identifies a container MBean. The `method` and `args` fields specify a method call to be performed on the base-level component associated with that container. The `invocationContext` conveys information that is common to all invocations performed through the same (base-level) object reference. It always includes information on whether the invocation target in an `EJBHome` or an `EJBObject`, and may also specify the id of a particular `EJBObject` instance[15]. Arbitrary invocation-specific information, typically related with non-functional aspects (e.g., security and transactions), is carried within the payload fields.

Note that the class `Invocation` is not serializable, as it has a non-serializable `java.lang.reflect.Method` field. `MarshalledInvocation`, an externalizable

[14] The `Invocation` objects are actually created by the invocation handlers of these dynamic proxies.

[15] There is an one-to-one relationship between deployed EJB components and container MBeans, but an EJB component usually has many `EJBObject` instances.

```
class Invocation {
    Object objectName;
    java.lang.reflect.Method method;
    Object[] args;
    InvocationContext invocationContext;
    java.util.Map payload;
    java.util.Map as_is_payload;     // marshalled "as is"
    java.util.Map transient_payload; // not sent to other VMs
    ... //  methods not shown
}
```

Fig. 8. Class that reifies method invocations

subclass of `Invocation`, serves the purpose of allowing reified invocations to be sent across virtual machine boundaries.

No interface along the reified invocation path (see Fig. 7) depends on base-level types: each element in that path provides a Java method or a management operation that takes an `Invocation` parameter. `Invocations` are eventually passed as parameters to an `invoke` operation exposed by all container MBeans:

```
mbeanServer.invoke(
    invocation.getObjectName(),              // target container
    "invoke",                                // operation name
    new Object[] { invocation },             // parameters
    new String[] {                           // parameter types
        "org.jboss.invocation.Invocation"
    }
);
```

Calls to a container's `invoke` operation go through the MBean server's `invoke` method, because the `invoke` operation belongs to the container's management interface.

4.2 Remote Invocation Architecture

A powerful and flexible architecture supports remote method invocations to EJB components deployed in JBoss. The invoker architecture is based on the following points:

- Even though EJB clients expect typed and application-specific interfaces, EJB containers expose a type-independent management operation (`invoke`), which acts as a meta-level gateway to their EJB components. An *invoker* makes this operation accessible to remote clients through some request/ response protocol, such as JRMP, IIOP, HTTP or SOAP.

```
interface Invoker extends javax.rmi.Remote {
    String getServerHostName();
    Object invoke(Invocation invocation);
}
```

Fig. 9. Generic invocation interface

- Client-side stubs (or client-side proxies) are dynamic proxy instances that convert calls to the typed interfaces seen by clients into `invoke` calls on remote invokers.
- Each client-side proxy has a serializable invocation handler that performs remote calls on a given invoker, over the protocol supported by the invoker.
- Client-side proxies and their invocation handlers are instantiated by the server and dynamically sent out to clients as serialized objects.

The pattern just outlined is independent of the request/response protocol supported by the invoker. Client-side knowledge of this protocol is confined within the invocation handlers that clients dynamically retrieve from the server along with serialized proxies.

Invokers. An invoker is a service MBean that acts as a protocol-specific gateway, at the meta level, to multiple EJB containers in the JBoss server. All invokers currently available in JBoss are deployable services implemented as standard MBeans. Every invoker exposes an `invoke` method to remote clients. This method takes an `Invocation` parameter and forwards the reified invocation to the container MBean specified by the invocation's `objectName` field.

Figure 9 shows the remote invocation interface exposed by the JRMP invoker, which makes its `invoke` method available to RMI/JRMP clients. Other invokers implement either this interface or very similar ones.

Client-Side Proxies. In order to access an EJB component deployed into a JBoss server, a client must have a reference to a client-side proxy that represents the component. Local calls to application-specific methods are translated by the client-side proxy into `invoke` calls on a remote invoker object. To perform this translation, the proxy — or, more precisely, its invocation handler — must *know* the remote invoker. The exact meaning of "knowing the remote invoker" depends on the protocol over which the proxy interacts with the remote invoker. In the case of a client-side proxy associated with a JRMP invoker, that phrase means "holding an RMI/JRMP reference to the JRMP invoker". For client-side proxies associated with other invokers, the same phrase takes other meanings, such as "knowing the HTTP invoker's URL", or "having a CORBA IOR that references the IIOP invoker".

Invoker Proxies. Everything that is protocol-specific within a client-side proxy is encapsulated within an *invoker proxy.* Regardless of the protocol it supports, each invocation handler holds a local reference to an invoker proxy that implements the `Invoker` interface shown in Fig. 9 [16]. The invoker proxy interacts with a remote invoker, sending `Invocation`s and receiving results over a given protocol. Invoker proxies provide a good level of homogeneity to all client-side proxies.

The invoker proxy for a given protocol is created at the server side and bound to a name in a well-known JNDI context. Since invoker proxies are externalizable, they can be sent out to clients along with serialized client-side proxies. Both the creation and the JNDI registration of a given protocol's invoker proxy (e.g., a `JRMPInvokerProxy`) are performed within the `create` operation of the invoker MBean for that protocol (e.g., within the `create` operation of the `JRMPInvoker` service component.)

Local Invocations. In-process calls between components deployed in the same server are optimally handled by a *local invoker* that avoids the cost of marshaling `Invocation`s. Local invocations go through client-side proxies and are reified like remote invocations, but in the local case a client-side proxy contains the local invoker itself, not an invoker proxy. Unlike the other invokers, which afford call-by-value semantics, the local invoker provides call-by-reference semantics.

The IIOP Case. For interoperability with CORBA clients written in other languages, IIOP is treated as a special case in JBoss. Even though we have implemented and tested an experimental IIOP invoker that strictly follows the "JBoss invoker pattern", this is *not* the IIOP invoker included in JBoss distributions.

Non-Java clients expect application-specific interfaces to be exposed via IIOP, because they use IDL-generated stubs. In other words, they send out IIOP requests whose operation fields contain application-specific verbs. The invoker pattern, however, leads to an IIOP invoker that implements an IDL interface similar to the Java interface in Fig. 9. Such an invoker could not possibly interoperate with CORBA clients written in other languages, as it would expect IIOP requests with the verb `invoke` in their operation fields. Rather than implementing the invoker pattern, the IIOP invoker included in JBoss follows the standard CORBA/IIOP approach, and hence it does not suffer from language interoperability problems.

Invoker Advantages. JBoss invokers present significant advantages over remote invocation architectures such as CORBA and Java RMI:

[16] For the moment, assume that there are no client-side interceptors interposed between the invocation handler and the invoker proxy (see Fig. 7). Interceptors will be discussed in Sect. 4.4.

- *Dynamic generation and retrieval of client-side proxies.* No application-specific stub classes have to be pre-installed in client machines.
- *Extensibility.* Multiple protocols can be simultaneously supported by various invokers and their invoker proxies. Support for new protocols can be added to a running server by dynamically deploying new invoker MBeans.
- *Multiple protocols per EJB.* An EJB component may receive remote invocations over different protocols, i.e., there may be a many-to-many relationship between container MBeans and invoker MBeans.
- *Multiple protocols per client.* Clients receive serialized proxies from other processes and use these proxies to issue remote method calls. Depending on the serialized proxies it receives, a client may employ multiple protocols to interact with various server-side components, without ever being aware of this fact.
- *Separation of concerns.* Invokers draw a very clear separation between middleware concerns inherent to distributed environments (e.g., protocol and fail-over strategy) and other kinds of concerns, either at the middleware level or at the application level.

Together, separation of concerns *and* dynamic retrieval of client-side proxies have far-reaching consequences. Configurable support to failover in clustered JBoss environments is one such consequence. It does not require any special arrangements at the client side, as failover is performed by client-side proxies dynamically retrieved from a JBoss server.

Comparison with CORBA. JBoss invokers afford separation of concerns at a much higher degree than a standard CORBA/IIOP approach. By putting application-specific verbs in a header field of IIOP requests, CORBA effectively forces remote interactions to take place at the base level, rather than at the meta level. We consider this a serious limitation of CORBA/IIOP.

4.3 Containers

When an EJB is deployed into JBoss, a container MBean is created to manage the EJB. Each reified EJB invocation is routed through a container, which provides its EJB with services such as instance pooling, instance caching, persistence, security, and transactions. The container MBean itself does not perform any of these services, it merely aggregates *container plug-ins* that do the real work. A container framework assigns specific responsibilities (bean instance pooling, bean instance caching, and management of bean persistence) to well-defined types of container plug-ins. Besides defining the interfaces that these plug-in types must implement, the framework also accepts a plug-in type — server-side interceptors — whose responsibilities are not specified in advance.

Note that Fig. 7 represents server-side interceptors *within* the EJB container. In JBoss, the abstract notion of an EJB container is realized by a container MBean, together with its set of container plug-ins. Instance pooling, instance caching, and persistence management plug-ins do not appear in that figure (we do not be discuss them in this paper), but they are logically encompassed by the EJB container as well.

JBoss containers are service components implemented as dynamic MBeans. They are created by the `EJBDeployer`, which reads container configurations from XML files.

Container Configurations. A container configuration specifies all information the `EJBDeployer` needs to create a container MBean, its plug-ins, and its interceptors. The configuration defines a Java class for every plug-in and every server-side interceptor, as well as values for configuration parameters. It also defines one or more kinds of client-side proxies that the container will export to EJB clients. For each kind of client-side proxy, it specifies the Java class of every client-side interceptor that will be included in client-side proxies, as well as the invoker MBean (that is, the protocol) that these proxies will use. See [24] for details.

Different kinds of EJBs require containers with different configurations. JBoss has a global configuration file that includes default container configurations for the standard kinds of EJBs (stateless session beans, stateful session beans, entity beans, and message-driven beans). The global configuration file also contains alternative configurations for these kinds of EJBs. A local configuration file, optionally included with a given EJB, may refer to an alternative configuration by its name, in order to specify some non-standard feature such as clustering. Moreover, local configuration files are not constrained to use pre-defined container configurations. A local configuration file may fully define a new container configuration, possibly specifying plug-in and interceptor classes included within the EJB deployment unit.

Generalized EJBs. Local container configurations effectively generalize the EJB model, allowing users to define EJB-like components suited to their needs. For instance, an user that does not need transactions or security can easily create a customized container configuration without transaction or security interceptors, which would otherwise perform superfluous tasks. Local configurations can also be employed to create containers that provide enhanced services. Customized containers can offer assertion capabilities [29] that verify whether component-based applications maintain certain critical properties or not. They can also provide OGSA[17] services in a grid computing environment. These are real examples of enhanced containers, created by JBoss users working on high-confidence systems at MITRE Corporation and by researchers working on the Globus Project, respectively.

[17] OGSA is the Open Grid Services Architecture [10] defined by the Global Grid Forum.

```
public abstract class Interceptor
        implements java.io.Externalizable {
    protected Interceptor nextInterceptor;
    public Interceptor setNext(final Interceptor interceptor) { ... }
    public Interceptor getNext() { ... }
    public void writeExternal(final ObjectOutput out) { ... }
    public void readExternal(final ObjectInput in) { ... }
    public abstract Object invoke(Invocation inv);
}
```

Fig. 10. Base class of client-side interceptors

4.4 Interceptors as Pluggable Aspects

Pluggable aspects, specified in configuration files written in XML, affect every EJB invocation performed at the base level. Each such aspect corresponds to an interceptor that acts directly upon reified invocations. Figure 7 shows two interceptor chains interposed across the reified invocation path: one at the client side, between the invocation handler and its invoker proxy, another at the server side, between a container MBean and its EJB component.

In aspect-oriented terminology, client-side proxies and containers provide *join points* in the invocation path, and interceptors implement *around advice* that runs at those join points. Each interceptor explicitly calls the next element in its chain, so it has complete control over whether the next element will be called or not.

Client-Side Interceptors. These interceptors inherit from the `Interceptor` class shown in Fig. 10. The field `nextInterceptor` and the methods `setNext` and `getNext` support singly-linked chains of interceptors. Class `Interceptor` is externalizable and provides default implementations of the externalization methods `writeExternal` and `readExternal`. Externalization is crucial for client-side interceptors, as it allows interceptor chains to be built at the server side and dynamically retrieved by clients, along with dynamic proxies. The remaining method, `invoke`, must be provided by concrete subclasses of `Interceptor`.

Client-side interceptors typically deal with aspects that involve some form of context propagation from the client to the server (e.g., transactions and security) or handle certain invocations whose processing can be completed at the client side (e.g., `getHandle`/`getHomeHandle` calls on `EJBObject`/`EJBHome` proxies).

Server-Side Interceptors. Unlike client-side interceptors, server-side interceptors do not need to be externalizable. Rather than inheriting from the externalizable class in Fig. 10, they implement a Java interface that resembles that class, but does not extend `java.io.Externalizable` or `java.io.Serializable`.

Server-side interceptor chains are typically longer than client-side ones. Besides dealing with transactions and security at the server side, they handle aspects such as logging, gathering of statistical data, entity instance creation, entity instance locking, detection of reentrant calls, and management of relationships between entities.

5 Project History

There have been four major versions of JBoss. The earliest[18] one (Feb. 1999) was still called EJBoss, a name soon abandoned for trademark reasons. EJBoss was an EJB server that introduced a novel feature — hot deployment — but still used a traditional and compilation-based approach to generate client stubs. JBoss 1.0 (Feb. 2000) was an innovative EJB server that employed a new technology — dynamic proxies — to avoid statically generated client stubs. In version 2.0 (Nov. 2000), JBoss was redesigned and rewritten as a complete J2EE implementation, modularly built around a JMX microkernel. JBoss 2.0 was already an extensible and reflective server that supported server-side interceptors and service MBeans, but not dynamic deployment of service MBeans. Even though invokers were already pluggable in version 2.0, they were container plug-ins, not service MBeans, and there was exactly one invoker (protocol) per EJB container. Version 3.0 (May 2002) featured dynamic deployment of service MBeans, dependency management, unified class loaders, client-side interceptors, and invokers as dynamically deployable MBeans. Multiple invokers per container started to be supported in JBoss 3.2 (first beta release available in Sep. 2002), the version described in this paper.

6 Ongoing and Future Work

Ongoing work includes EJB 2.1 compliance, performance optimizations, and, most importantly, an aspect-oriented programming framework that will allow class, method, field, and constructor pointcuts to be dynamically attached to any Java object. In the near future, this framework should provide the basis for an extended version of the JMX infrastructure within JBoss, as well as for a new implementation of the meta-level architecture described in Sect. 4. Moreover, we expect that close integration with an aspect-oriented programming framework will help JBoss on the move from a generalized EJB model to yet more flexible models for application components.

Another area for future work is on extensions to the metaobject protocol supported by generalized EJB containers. Sensible changes to meta-level elements of the generalized EJB architecture should be allowed in a yet more dynamic way. Management clients, such as the JBoss console, currently can update meta-level

[18] Except when explicitly stated otherwise, this paragraph informs "final release" dates. Each final release was preceded by a series of alpha and beta releases that started to appear many months earlier.

attributes and invoke meta-level operations (e.g., a management operation that flushes the instance cache of an entity container). In order to update the container configuration used by some application component, however, they must redeploy the component. (More precisely: they must deploy a new version of the component package, with changes in its local configuration file.) By interacting with the JBoss console, one should be able to add a server-side interceptor or a new protocol (invoker) to a component deployed into a running server, without redeploying the component.

7 Related Work

Support to JMX started to appear in commercial J2EE servers. These systems employ JMX merely as a means to instrument selected parts of the application server and make them manageable through JMX-compliant clients. As far as we know, none of the commercial systems uses JMX as a reflective microkernel architecture at the very basis of the whole server.

A recent paper [3] discusses the performance of EJB applications. It presents a comparative evaluation of the performance of certain applications deployed both on JBoss and on a less flexible server, which relies on compilation to generate stubs and is not built around a reflective microkernel.

FlexiNet [13] is a Java middleware system that exploits reflective techniques in order to support flexible remote method invocation paths (*protocol stacks*, in FlexiNet terminology). Such an invocation path consists of a dynamically generated (and very thin) client-side stub, followed by a client-side chain of metaobjects, which interacts with a server-side chain of metaobjects that ends on the target object. Invocation targets are identified by *names* created by *generators* and resolved by matching *resolvers*. When a name is created, its generator also creates the server-side half of an invocation path. The resolution of the name entails the construction of the client-side half of that path. FlexiNet includes generator-resolver pairs (called *binders*) for various protocols.

Unlike JBoss, FlexiNet does not follow Java RMI conventions to decide whether a method parameter will be passed by value or by reference. Rather than examining the type of the *actual* parameter, it looks at the declared type of the *formal* parameter. The parameter will be passed by reference if its declared type is an interface, and will be passed by value otherwise[19]. Similarly, when passing by value an object with fields that refer to other objects, FlexiNet looks at the declared types of these fields. As standard Java serialization does not provide this semantics, FlexiNet implements its own serialization mechanism. It also implements its own factory of dynamic stubs[20], which differ from the dynamic proxies employed by JBoss in that they cannot implement multiple

[19] This convention is incompatible with the widely accepted advice that one should favor the use of interfaces (rather than classes) as parameter types. [1]

[20] The development of FlexiNet preceded the inclusion of support to dynamic proxies in the Java reflection API.

interfaces. In comparison with FlexiNet, the JBoss remote invocation architecture is considerably simpler, due to its usage of standard Java features (dynamic proxies and object serialization). The serialized form of a dynamic proxy, along with its invocation handler, is the JBoss counterpart of a FlexiNet name. JBoss employs standard Java deserialization as a universal (protocol-independent) "resolver" that converts "names" into client-side stubs.

Yasmin [6,7] is a component-based architecture designed with emphasis on distributed applications for network management. The Yasmin model is not Java-based, but it supports hot-deployable components ("*droplets*") that resemble our deployable MBeans. A major difference, however, is that Yasmin does not address dependence management issues.

OpenCOM [4] is a lightweight component model upon which an adaptive ORB has been implemented. As an in-process model built atop a subset of Microsoft's COM, OpenCOM appears more suitable for very fine-grained components than the JBoss service component model. It supports dependence management, reconfiguration, and method call interception. Nevertheless, OpenCOM does not address deployment issues, nor does it support dynamic loading of component classes from remote locations.

Other component models have been proposed for systems software. Some of these models, which bear less similarity with the JBoss/JMX model than Yasmin and OpenCOM, are discussed in [4].

8 Concluding Remarks

JBoss demonstrates that application servers can be built out of dynamically deployed components that provide middleware services to application components. At the architectural level, its novel contributions include the pioneering usage of JMX as a reflective microkernel architecture, a JMX-based component model with support to dynamic deployment and dependence management, and a meta-level architecture for generalized EJBs. At the implementation level, JBoss innovations include a class loading model that facilitates sharing of classes across deployment units, as well as an "invoker pattern" that relies on serialization of dynamic proxies created at the server side in order to support a general remote method invocation service over multiple protocols.

Researchers have recently advocated "working toward standards for reflective middleware" [17]. The JBoss experience suggests that reflective models based on JMX should be seriously considered as candidates to standardization not only within the network and systems management field, but in the more general Java middleware arena.

Acknowledgements

The JBoss open-source server was designed and implemented by an international team led by Marc Fleury. At the time of this writing, the team has 72 members geographically dispersed across five continents. An up-to-date listing of team members is available at `http://sf.net/projects/jboss`. Past and present contributors are listed in `http://www.jboss.org/team.jsp`. Special credit is due to Rickard Öberg, the main designer of JBoss 2.0. His sound decisions are still a major driving force for the project.

The authors wrote this paper as rapporteurs for a group of kernel JBoss contributors that includes Scott Stark, Bill Burke, David Jencks, Sacha Labourey, Juha Lindfors, Dain Sundstrom, Adrian Brock, Jason Dillon, Christoph Jung, Hiram Chirino, and the authors themselves.

References

1. J. Bloch. *Effective Java.* The Java Series. Addison-Wesley, 2001.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.
3. E. Checchet, J. Marguerite, and W. Zwanepoel. Performance and scalability of EJB applications. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, 2002.
4. M. Clarke, G. S. Blair, G. Coulson, and N. Parlavantzas. An efficient component model for the construction of adaptive middleware. In *Middleware 2001 — IFIP/ACM International Conference on Distributed Systems Platforms*, volume 2218 of *LNCS*, pages 160–178. Springer-Verlag, 2001.
5. M. Dahm. Byte code engineering with the BCEL API. Technical Report B–17–98, Freie Universität Berlin — Institut für Informatik, 1998.
6. L. Deri. *A Component-Based Architecture for Open, Independently Extensible Distributed Systems.* PhD thesis, University of Berne, Switzerland, 1997.
7. L. Deri. Yasmin: A component-based architecture for software applications. In *8th International Workshop on Software Technology and Engineering Practice (STEP'97)*, pages 4–12, London, July 1997. IEEE Computer Society. Also published as IBM Research Report RZ 2899.
8. T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, October 2001.
9. J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of the 4th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'89)*, pages 317–326, 1989.
10. I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.
11. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns — Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
12. S. D. Halloway. *Component Development for the Java Platform.* Addison-Wesley, 2002.
13. R. Hayton and ANSA Team. FlexiNet Architecture. ANSA Architecture Report, Citrix Systems Ltd., Cambridge, UK, February 1999. Available in the ANSA web site (`http://www.ansa.co.uk`).
14. T. Jewell. EJB 2 and J2EE packaging, part II, July 2001. Available in O'Reilly's OnJava web site (`http://www.onjava.com`).
15. C. G. Jung. `java.lang.ClassLoader.loadClassInternal(String)` is too restrictive. Bug report submitted to `java.sun.com` (bug id 4670071), April 2002.

16. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.
17. F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, June 2002.
18. S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 36–44, 1998.
19. J. Lindfors, M. Fleury, and The JBoss Group. *JMX: Managing J2EE with Java Management Extensions.* SAMS, 2002.
20. J. P. Martin-Flatin, S. Znaty, and J. P. Hubaux. A survey of distributed enterprise network and systems management paradigms. *Journal of Network and Systems Management*, 7(1):9–26, 1999.
21. B. Peterson. Understanding J2EE application server class loading architectures, May 2002. Available in TheServerSide.com (`http://www.theserverside.com`).
22. D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects.* Wiley, 2000.
23. SourceForge.net. Monthly download statistics for JBoss, December 2002. `http://sf.net/project/stats/index.php?report=months&group_id=22866`.
24. S. Stark and The JBoss Group. *JBoss Administration and Development, Edition 2.* JBoss Group, 2002.
25. Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. `http://java.sun.com/ejb/`.
26. Sun Microsystems. *Java 2 Platform Enterprise Edition Spec., v1.3*, 2001. `http://java.sun.com/j2ee/`.
27. Sun Microsystems. *Java Management Extensions — Instrumentation and Agent Specification, v1.1*, 2002. `http://java.sun.com/jmx/`.
28. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.
29. G. J. Vecellio, W. M. Thomas, and R. M. Sanders. Containers for predictable behavior of component-based software. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering: Benchmarks for Predictable Assembly*, Orlando, Florida, USA, 2002. Carnegie-Mellon Software Engineering Institute.