Dynamic Deployment of IIOP-Enabled Components in the JBoss Server

Francisco Reverbel[1], Bill Burke[2], and Marc Fleury[2]

[1] Department of Computer Science, University of São Paulo
reverbel@ime.usp.br

[2] JBoss, Inc.
{bill,marc}@jboss.org

Abstract. JBoss is an extensible Java application server that affords remote access to EJB components via multiple protocols. Its IIOP module supports IIOP-enabled EJBs, which are accessible both to RMI/IIOP clients written in Java and to CORBA clients written in various languages. While other systems use compilation-based approaches to generate IIOP stubs and skeletons, JBoss employs reflective techniques to avoid extra compilation steps and support on-the-fly deployment.
CORBA/IIOP is a dynamic feature of JBoss in two senses: (*i*) the IIOP module can be dynamically deployed into a running server, and (*ii*) IIOP-enabled EJBs are dynamically deployable components themselves. This paper presents the design of the IIOP module and describes the actions that module takes at EJB deployment time, including the creation of POAs, the instantiation of CORBA servants to implement IDL interfaces not known in advance, and the dynamic generation of IIOP stub classes made available to Java clients via HTTP.

1 Introduction

JBoss [10] is an extensible, reflective, and dynamically reconfigurable Java application server that supports two general kinds of software components: *application components*, which correspond to server-side parts of distributed applications, and *middleware components*, which provide middleware services to application components. Both kinds of components can be dynamically deployed into a running server. Users can deploy components either by interacting with the server through a (possibly remote) management client or simply by dropping deployment units in a well known directory of the server machine.

Nearly all the "application server functionality" of JBoss is modularly provided by a set of middleware components deployed on a minimal server. While middleware components employ a JBoss-specific extension of the JMX component model [17], application components follow Java 2 Enterprise Edition (J2EE) standards [19]. An important subset of application components is based on the Enterprise JavaBeans (EJB) architecture [18], a model for business components ("enterprise beans") whose methods can be invoked either by remote clients or by local clients.

EJB-compliant application servers may support various protocols, but they must support IIOP as the interoperability protocol for remote method invocations on enterprise beans. RMI over IIOP affords interoperability between EJB components deployed in application servers provided by different vendors. It makes enterprise bean methods available both to RMI/IIOP clients written in Java and to CORBA clients written in various languages, including Java and C++. Even though IIOP-enabled EJBs are not equivalent to the CORBA components [14] defined by the OMG, they can actually be regarded as another kind of "CORBA component."

JBoss supports IIOP in a *dynamic* way. Here the word "dynamic" means two things: (*i*) IIOP support is a feature that can be dynamically added to a running server, and (*ii*) IIOP-enabled EJBs are dynamically deployable components themselves. At the root of the latter point there is a major difference between our "CORBA components" and existing Java implementations of the CORBA Component Model (CCM). In current CCM implementations [8, 15], component deployment involves an additional compilation step, as these systems use compilation-based approaches to generate IIOP stubs and skeletons. Our work, on the other hand, avoids extra compilation steps by employing reflective techniques. It allows IIOP-enabled EJBs to be deployed into a running server simply by dropping EJB-JAR files in the server's deployment directory. A change to a JBoss-specific deployment descriptor is all that is needed to convert a non-IIOP-enabled deployment unit (which contains neither IIOP stubs nor skeletons) into an IIOP-enabled one. There are no additional steps for stub or skeleton generation.

This paper is organized as follows: Section 2 contains background material, mostly extracted from [10], Section 3 presents the middleware components that support CORBA/IIOP, Section 4 describes the proxy factories that generate remote references to EJBs, Section 5 discusses design and implementation issues, Section 6 reviews related work, and Section 7 presents our concluding remarks.

2 JBoss Background

2.1 JMX

Java Management Extensions (JMX) [17] is an architecture for dynamic management of resources (applications, systems, or network devices) distributed across a network. It provides a lightweight environment in which components — as well as their class definitions — can be dynamically loaded and updated.

JMX components (*MBeans*) are Java objects that conform to certain conventions and expose a *management interface* to their clients. A Java virtual machine that contains JMX components must contain also an *MBean server*, which provides a local registry for 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 on an MBean, a local client (typically another MBean) issues a generic `invoke` call on the MBean server, passing the target MBean's object name as an argument. The client needs no information on the MBean's Java class, nor does it need information on the Java interfaces the MBean implements. 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.

JBoss uses JMX as a realization of the microkernel architectural pattern [6], to provide a minimal kernel (the MBean server) that serves as software bus for extensions (MBeans), possibly developed by independent parties. The MBean server decouples components from their clients, allowing MBeans to adapt, and their management interfaces to evolve, while their clients are active.

2.2 Service Components

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.

Service MBeans (also called *service components*) are JMX 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*, an XML file that conveys information needed at deployment time.

Service components plugged into a JMX-based "server spine" provide most of the "application server functionality" offered by JBoss. They 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. Service components also implement important features not specified by J2EE, such as clustering and fail-over.

2.3 Meta-Level Architecture for Generalized EJBs

The conceptual definition of the EJB architecture [18] 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 components;
- a uniform model for reifying base-level method invocations;
- usage of dynamic proxies as the basis for a remote invocation model that supports multiple protocols;
- a variant of the interceptor pattern [16], used as an aspect-oriented programming [11] technique.

In this section we restrict ourselves to the case of EJB clients that do not use IIOP to interact with remote EJB components. Fig. 1 shows (in a non-IIOP scenario) the elements of the meta-level architecture implemented by JBoss. The base level consists of EJB components and their clients. Accordingly, we 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.

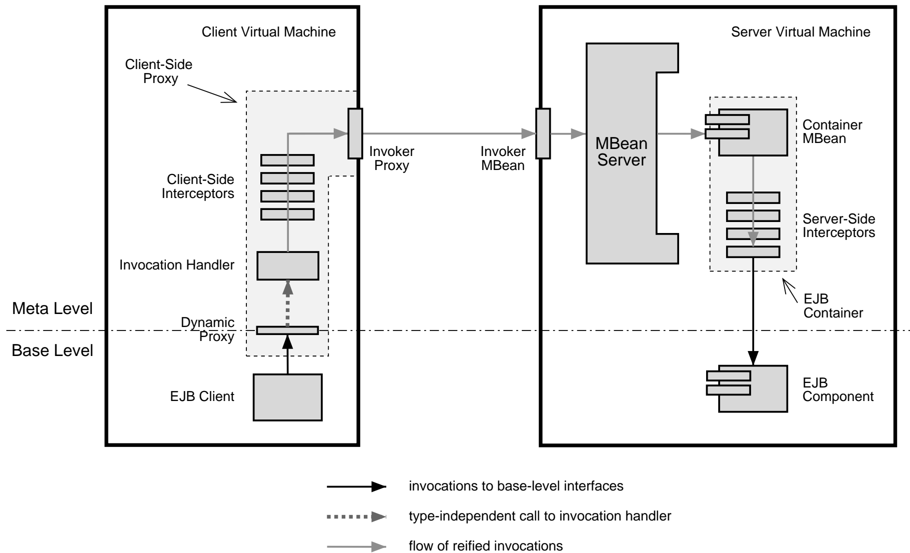

Fig. 1. Meta-level architecture for EJB: the case of a non-IIOP client

Interactions between base-level components follow a variant of the message reification model [9]. Inter-component method invocations performed at the base level are reified by special `Invocation` objects. Dynamic proxies receive all EJB invocations executed by non-IIOP clients (which may be EJBs themselves) and shift those invocations up to the meta level, by transparently converting them into `Invocation` objects.

The gray arrows in Fig. 1 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 2 lists some of 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 is an `EJBHome` or an `EJBObject`, and may also specify the id of a particular `EJBObject` instance.

```
class Invocation {
    Object objectName;
    java.lang.reflect.Method method;
    Object[] args;
    InvocationContext invocationContext;
    ...
}
```

Fig. 2. Class that reifies method invocations

2.4 Remote Invocation Architecture for Non-IIOP Clients

Even though EJB clients expect typed and application-specific interfaces, EJB containers expose the generic management operation `invoke`, which takes an `Invocation` parameter. This operation plays the role of meta-level gateway to the EJBs deployed within a JBoss server. A flexible architecture supports remote invocations to EJB components by exposing the `invoke` operation through various protocols:

- An *invoker* makes the container's `invoke` operation accessible to remote clients through some request/response protocol, such as JRMP, HTTP or SOAP.
- 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 to multiple EJB containers in the JBoss server. All invokers currently available in JBoss are deployable services implemented as standard MBeans. Each non-IIOP 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.

Fig. 3 shows the remote invocation interface exposed by the JRMP invoker, which makes its `invoke` method available to RMI/JRMP clients. Other non-IIOP invokers implement either this interface or very similar ones.

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

Fig. 3. Generic invocation interface

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 (in a clustered JBoss environment) "holding a collection of references to target invokers distributed across cluster nodes."

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. 3. The invoker proxy interacts with a remote invoker, sending `Invocation`s and receiving results over a given protocol. Invoker proxies are created at the server side (as protocol-specific singletons) and sent out to clients along with serialized client-side proxies. They provide a good level of homogeneity to all client-side proxies.

3 The IIOP Module

Three deployable MBeans support CORBA and IIOP as a dynamically deployable feature: `CorbaORBService`, `CorbaNamingService`, and `IIOPInvoker`. They are packaged together as an "IIOP module", also known as JBoss/IIOP.

3.1 IIOP Engine

The `CorbaORBService` MBean is a thin wrapper around a third-party IIOP engine, which can be any Java ORB compliant with CORBA 2.3 or later. JacORB [4, 13], a free Java implementation of the CORBA standard, is the default IIOP engine included in the JBoss distribution. The IIOP engine is pluggable and may be replaced by users: attributes of the `CorbaORBService` MBean determine which ORB will be used. The values of these attributes are configurable via XML elements in the IIOP module's service descriptor.

3.2 CORBA Naming Service

IIOP-enabled EJBs are registered with the naming service made available by the `CorbaNamingService` MBean. Rather than allowing an external naming service to be plugged in via MBean attributes, this MBean provides an in-process CORBA naming service. It implements this service by reusing (through inheritance) code from the JacORB naming service. Users that want an external CORBA naming service have the option of replacing the `CorbaNamingService` MBean by a simpler one, which merely makes an external naming context available for EJB registration.

3.3 IIOP Invoker

The `IIOPInvoker` MBean differs from all other JBoss invokers in that it does not follow the pattern outlined in Section 2.4. 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 `IIOPInvoker` 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. 3. 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 `IIOPInvoker` included in JBoss follows the standard IIOP approach (albeit in a more dynamic way than most CORBA servers), and hence it does not suffer from language interoperability problems.

The `IIOPInvoker` maintains a collection of Portable Object Adapters (POAs) and a collection of CORBA servants. Fig. 4 shows how it fits into the JBoss meta-level architecture. The EJB client can be either an RMI/IIOP client written in Java, or a plain CORBA client, possibly written in another language. By performing a method invocation on an IIOP stub, the client causes an IIOP request to be sent out through the client-side ORB. The server-side ORB forwards the request to a POA, which issues an up-call to a CORBA servant. (Both the POA and the servant are logically contained in the `IIOPInvoker`.) The servant then converts the request into an `Invocation` object and routes the reified invocation through the container MBean associated with the target EJB.

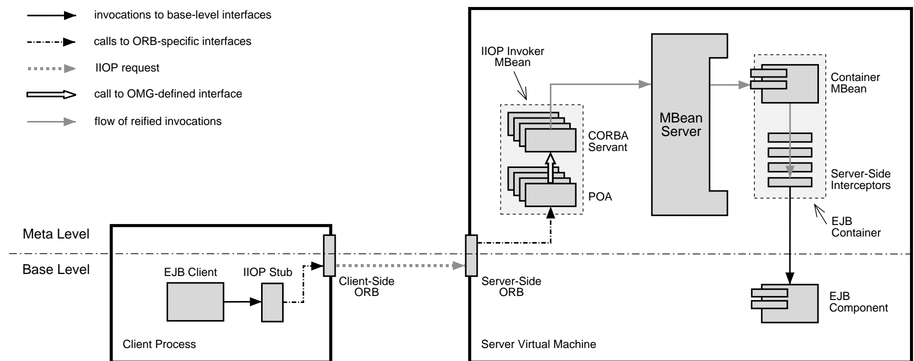

Fig. 4. Meta-level architecture for EJB: the case of an IIOP client

Note that the client/server interaction appears at different levels in the non-IIOP case (Fig. 1) and in the IIOP case (Fig. 4). The interaction takes place at the meta level in non-IIOP case, because non-IIOP invokers expose a meta-level interface (the `Invoker` interface) to remote clients. The `IIOPInvoker`, on the other hand, affords remote access to base-level EJB interfaces. More precisely, it gives remote clients access to IDL counterparts (per the Java to IDL mapping) of application-specific EJB interfaces. This is a CORBA/IIOP interoperability requirement.

4 Proxy Factories

An *EJB reference* is either a reference to an `EJBObject` or a reference to an `EJBHome`. EJB containers have the responsibility of creating such references, which in various situations they pass to EJB clients (e.g., when a client invokes a finder method on an `EJBHome`) or to EJB implementations (e.g., when a bean implementation invokes `getEJBObject()` on an `EJBContext`).

JBoss supports two general kinds of EJB references: (*i*) Java references to client-side stubs implemented as dynamic proxy instances, which are sent out to

other virtual machines as serialized Java objects, and (*ii*) CORBA references, passed across process boundaries in the IOR format standardized by the OMG. In either case, a container MBean delegates to a *proxy factory* the task of creating EJB references.

Proxy factories implement the interface partly shown in Fig. 5. Three proxy factory classes currently implement that interface:

- `ProxyFactory`. Instances of this class create dynamic stubs (client-side stubs implemented as dynamic proxies) that talk to remote invokers over various protocols.
- `ProxyFactoryHA`. Instances of this class create dynamic stubs used in clustered JBoss environments. (The suffix "HA" stands for "high availability.")
- `IORFactory`. Instances of this class create CORBA references.

```
public interface EJBProxyFactory extends ContainerPlugin {
    ...
    Object getEJBHome();
    Object getStatelessSessionEJBObject();
    Object getStatefulSessionEJBObject(Object sessionId);
    Object getEntityEJBObject(Object primaryKey);
    Collection getEntityCollection(Collection prymaryKeys);
}
```

Fig. 5. Proxy factory interface

4.1 Relationship between Proxy Factories and Invokers

Each proxy factory is associated with an invoker MBean. Distinct `ProxyFactory` instances (or `ProxyFactoryHA` instances) may be bound to different invokers, e.g.:

- A `ProxyFactory` associated with the `JRMPInvoker` creates dynamic stubs that interact with the JBoss server via JRMP.
- A `ProxyFactory` associated with the `HTTPInvoker` creates dynamic stubs that interact with the JBoss server via HTTP.

`IORFactory` instances can only be bound to the `IIOPInvoker`; they are currently the only kind of proxy factory that can be associated with the `IIOPInvoker`.

4.2 Relationship between Proxy Factories and Containers

An EJB component deployed in JBoss has its own container, i.e., there is an one-to-one relationship between deployed EJB components and container MBeans.

Each container MBean owns a collection of proxy factories: it has a proxy factory instance for every protocol supported by its EJB component. Since these proxy factories are also associated with invoker MBeans, they define a many-to-many relationship between container MBeans and invoker MBeans.

Invoker MBeans are shared among containers (there is one invoker per protocol), but proxy factories are not. A proxy factory instance knows how to create dynamic stubs or IORs that correspond to the `EJBHome` or to an `EJBObject` implemented by a given container. Information on the identity of that container is included in every dynamic stub created by a `ProxyFactory`. The JNDI name of the EJB deployed into that container is embedded within the object key of every IOR created by an `IORFactory`.

4.3 Proxy Factory Configurations

At deployment time, the EJB deployer (an MBean that handles the deployment of EJB components) reads container configurations from XML files and creates containers. A *container configuration* has all the information the EJB deployer needs to create a container MBean, its plug-ins, and its interceptors. This includes information on the container's proxy factories, which are a special case of container plug-in. For each kind of client-side proxy that a container will export to EJB clients, the container configuration specifies a *proxy factory configuration*. XML elements in the proxy factory configuration specify the proxy factory class (a Java class) and the invoker MBean (that is, the protocol) to be used by the exported proxies, as well as additional parameters, which depend on the proxy factory class.

In the case of non-IIOP access to EJBs, XML elements in the proxy factory configuration fully specify the chain of client-side interceptors (see Fig. 1) to be included in every dynamic stub created by that factory. No similar elements exist in the configuration of a proxy factory for IIOP access to EJBs (an `IORFactory`), because interceptors instantiated at the server side would not make sense to non-Java clients. Serialized Java objects received from the server would be meaningless to CORBA clients written in other languages.

Global Configuration. 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. The default container configurations support remote access to EJBs by RMI/JRMP clients; alternative configurations support all other scenarios: remote access to EJBs by RMI/IIOP and CORBA clients, clustered session beans, etc. For each such scenario, the global configuration file has a proxy factory configuration, which container configurations may reference by name.

Local Configurations. A JBoss-specific deployment descriptor, optionally included with a given EJB, may refer to an alternative container configuration

by its name, either to specify other protocol such as IIOP, or to use some non-standard feature such as clustering. Moreover, those deployment descriptors are not constrained to use container configurations defined in the global configuration file. A JBoss-specific descriptor may fully define a new container configuration, possibly specifying plug-in and interceptor classes included within the EJB deployment unit. More frequently, however, it will refer to a predefined container configuration and override some elements of that configuration.

The JBoss-specific deployment descriptor in an EJB may use a predefined container configuration and enhance it with additional proxy factory configurations, possibly also taken from the global configuration file. This way one can easily specify that an EJB should be simultaneously accessible via multiple protocols (e.g., RMI/JRMP, HTTP and IIOP).

5 IIOP Invoker and IOR Factory Internals

Recall that the EJB deployer creates an `IORFactory` whenever it deploys an IIOP-enabled EJB. All IIOP-related actions taken at EJB deployment time are performed within the initialization of the `IORFactory`. These actions include the instantiation of home and bean servants, the creation of POAs, the registration of the `EJBHome` in a JNDI context, and the creation of a specialized class loader that lazily generates class definitions for RMI/IIOP stub classes.

5.1 CORBA Servants

The `IIOPInvoker` has two CORBA servants per IIOP-enabled EJB deployed in JBoss: a *home servant*, which handles invocations on the `EJBHome`, and a *bean servant*, which handles invocations on all `EJBObject` instances of the IIOP-enabled EJB. What makes these servants interesting is that they must implement IDL interfaces not know in advance. A possible approach would instantiate EJB servants from classes created at deployment time. Another approach would rely on a dynamic (generic) server-side interface. To make deployment lighter, JBoss/IIOP follows the second approach.

CORBA standardizes two dynamic interfaces for server-side request dispatching: the dynamic skeleton interface (DSI), defined in IDL and mapped to various implementation languages, and the stream-based ORB API, specified only for the Java case. Both are equally powerful, but the DSI requires all operation parameters and results to be wrapped into CORBA `Any`s. To avoid this extra cost, JBoss/IIOP uses the stream-based API.

The CORBA servants depicted in Fig. 4 are stream-based dynamic skeletons that receive generic (type-independent) requests from POAs and convert these requests into `Invocation` objects, which they forward (through the MBean server) to container MBeans. Every servant knows the object name of the container MBean to which it should forward reified invocations. Moreover, each servant has marshalling knowledge specific to the IDL interface it implements.

Within a servant, marshalling knowledge takes the form of a map from IDL operation names to `SkeletonStrategy` objects. The `SkeletonStrategy` for a given operation knows how to read the sequence of operation parameters from an input stream, how to write into an output stream the return value of the operation, and how to write into an output stream any exception raised by the operation. It has an array of reader objects (instances of auxiliary classes such as `LongReader`, `StringReader`, `CorbaObjectReader`, ...) for the operation parameters, an instance of a writer class (e.g, a `LongWriter`) for the operation result, and an `ExceptionWriter` for each exception that the operation may raise.

As a side note: dynamic deployment of IIOP-enabled components appears to be a new use case for dynamic server-side interfaces. The DSI was introduced in CORBA to allow the construction of interdomain bridges[3]. When Java ORBs started to use the DSI as a portability layer for IDL-generated skeletons, a more efficient portability layer — the stream-based ORB API — was defined. To the best of our knowledge, previous usage of dynamic server-side interfaces was restricted to these two scenarios (interdomain bridges and portability layer for IDL-generated skeletons).

5.2 POA Usage

JBoss/IIOP generates CORBA references with the following lifetimes: references to session bean instances are transient, references to entity bean instances and to `EJBHome`s are persistent. (This choice of reference lifetimes is quite natural, albeit not mandated by EJB specification. Moreover, it allows EJB handles and home handles to be implemented as thin wrappers around IORs.) Accordingly, JBoss/IIOP registers session bean servants with POAs that have the `TRANSIENT` liefespan policy, while it registers entity bean and home servants with POAs that have the `PERSISTENT` lifespan policy.

An XML element in the configuration of an IIOP proxy factory (`IORFactory`) specifies either *per-servant* or *shared* POAs. In the per-servant case, there are two POAs per deployed EJB: one for the home servant (a `PERSISTENT` POA), the other for the bean servant (either a `TRANSIENT` POA or a `PERSISTENT` POA, depending on the kind of EJB). This pair of POAs is created at deployment time. In the shared case, no POAs are created at deployment time. All deployed EJBs share a pair of POAs: a `PERSISTENT` POA dispatches requests to home and entity bean servants, a `TRANSIENT` POA dispatches requests to session bean servants.

The default `IORFactory` configuration specifies per-servant POAs, which are preferred because they make it possible to specify tagged components (such as codebase components for stub downloading) that should be added to specific IORs. The tagged components included in the IORs created by a given POA are determined by policy objects associated with that POA. The per-servant

[3] Such a bridge must implement CORBA objects that act as proxies for objects in another domain, with no compile-time knowledge of the interfaces of those CORBA objects.

configuration thus allows the specification of IOR components on a per-EJB basis.

5.3 Registration of EJBHomes

At deployment time, the EJBHome of an IIOP-enabled EJB is registered with the in-process CORBA naming service. It may also be optionally registered with JNP (Java Naming Provider), a JBoss-specific naming service that implements the JNDI APIs.

5.4 Lazy Generation of RMI/IIOP Stub Classes

An IORFactory can be optionally[4] associated with a *web class loader* that generates RMI/IIOP stub classes in a lazy way. The web class loader is an instance of WebCL, a subclass of URLClassLoader. Besides performing on-the-fly generation of stub classes, WebCL has a method getBytes, which returns a byte code array given a Class instance.

Recall that RMI/IIOP stub classes are named by appending the suffix "_Stub" to names of remote Java interfaces. When a WebCL instance is asked to load the class Foo_Stub, it performs Java introspection on interface Foo, applies the Java to IDL mapping on Foo, and uses code generation techniques to generate a byte code array with the class file for Foo_Stub, from which it obtains a Class instance. The WebCL instance keeps the byte code array in a hash map and returns this array whenever it is asked (via getBytes) for the byte code definition of the Foo_Stub class.

JBoss includes an in-process web server that allows remote clients to dynamically download class files via HTTP. A client downloads a class file by issuing an HTTP request for a resource whose name has two parts: (*i*) the id of a WebCL instance that can load the class, and (*ii*) the full name of the requested class file. When the web server receives such a request, it uses the WebCL both to obtain a Class object and to retrieve the class file given the Class object. RMI/IIOP stubs are therefore generated in a lazy way, as the web server receives requests from RMI/IIOP clients. If the server receives no download requests for the stub class associated with some EJB interface (perhaps because the EJB does not have any RMI/IIOP clients, but only CORBA clients), then no byte code generation is ever performed for that stub class.

In the default IORFactory configuration, every IORFactory has its own WebCL instance. (In other words, there is a web class loader per IIOP-enabled EJB.) Moreover, an IORFactory includes information on its web class loader in each IOR it creates. Such an IOR has a tagged component that specifies a codebase URL for RMI/IIOP stub downloading. The path component in this URL identifies the web class loader of the IORFactory that created the IOR.

[4] The usage of a web class loader is specified by an XML element in the proxy factory configuration.

6 Related Work

The JBoss meta-level architecture resembles FlexiNet [12], a Java middleware system that exploits reflective techniques in order to support flexible remote method invocation paths and multiple protocols, including IIOP. However, FlexiNet does not define an application component model, nor does it address component deployment issues.

OpenCOM [7] 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.

Research prototypes have recently made significant advances at addressing dynamic deployment issues and supporting multiple protocols in component-based systems [1, 3, 5]. Unlike JBoss, these systems generally employ non-standard architectures and support component models not as well-established as EJB.

Very little has been published about the internal architecture of commercial J2EE servers. While JBoss employs reflective techniques, most commercial servers use compilation-based approaches. Nonetheless, the JBoss meta-level architecture appears to have important features in common with IONA's ART (Adaptive Runtime Technology) framework, which relies on the chain of responsibility pattern in order to support different transports, protocols, and interceptors, as well as different kinds of containers [20]. IONA uses ART as the basis for various middleware products, including a J2EE server.

Facilities for agile deployment of EJBs were absent from commercial J2EE servers until recently. In early J2EE products, the EJB deployment process even included a server restart. After JBoss introduced hot deployment of EJB components, this feature found its way into commercial servers, albeit with some restrictions. For example, BEA's WebLogic 8.1 supports hot deployment of EJBs, but does not recommend its usage in production environments [2]. Nearly all commercial offerings require vendor-specific EJB container or CORBA servant classes to be statically generated as part of the deployment process. In the case of IIOP-enabled EJBs, all other servers still require extra compilation steps for stub and skeleton generation. JBoss supports dynamic deployment of IIOP-enabled EJBs in a much stronger sense: a running server accepts IIOP-enabled deployment units that contain no JBoss-specific classes (such as EJB container or CORBA servant classes), no IIOP skeletons, and no IIOP stubs.

7 Concluding Remarks

Flexibility, developer friendliness, and ease of use are crucially important qualities in an application server. In fact, they have been frequently mentioned as

reasons for the popularity of JBoss. The ability of deploying components on-the-fly has a very strong influence on those qualities. The absence of additional compilation steps, such as IDL translation, is also a significant positive factor for developer friendliness.

One of the main design goals of JBoss/IIOP was to support IIOP-enabled components without sacrificing the levels of flexibility, developer friendliness and ease of use that JBoss had already reached in the non-IIOP case. Toward this end, we have taken a reflective approach to CORBA/IIOP, and particularly to IIOP-enabled EJB components. This paper presented in detail the design and the implementation of a set of middleware components that strongly relies on reflection to meet the goal of making EJB deployment easy. Emphasis was placed on dynamic deployment issues, including the instantiation of CORBA servants for IDL interfaces not known in advance, the creation of POAs, and the lazy generation of RMI/IIOP stubs. Those issues were discussed in the context of a J2EE application server, but we believe that our techniques are equally applicable to CCM servers and to other highly dynamic component environments, in areas such as mobile computing and grid computing.

Acknowledgments

We thank Ole Husgaard, who implemented the mapping from RMI types to IIOP types used in JBoss/IIOP. We also thank Gerald Brose, for creating JacORB, and André Spiegel, for his work on valuetype support in JacORB.

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 95 members geographically dispersed across five continents. An up-to-date listing of team members is available at `http://sf.net/projects/jboss`.

References

[1] D. Balek and F. Plasil. Software connectors and their role in component deployment. In *Proceedings of DAIS'01*, Krakow, September 2001. Kluwer.

[2] BEA Systems. WebLogic Server 8.1 Documentation, 2003.

[3] I. Ben-Shaul, O. Holder, and B. Lavva. Dynamic adaptation and deployment of distributed components in Hadas. *IEEE Transactions on Software Engineering*, 27(9):769–787, 2001.

[4] G. Brose. JacORB: Implementation and design of a Java ORB. In *Proceedings of DAIS'97*, pages 143–154. Chapman & Hall, 1997.

[5] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, 2002.

[6] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[7] 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.

[8] EJCCM web site, 2003. http://www.cpi.com/ejccm/.
[9] J. Ferber. Computational reflection in class-based object-oriented languages. In *Proceedings of OOPSLA'89*, pages 317–326, 1989.
[10] M. Fleury and F. Reverbel. The JBoss extensible server. In *Middleware 2003 — ACM/IFIP/USENIX International Middleware Conference*, volume 2672 of *LNCS*, pages 344–373. Springer-Verlag, 2003.
[11] G. Kiczales *et al*. Aspect-oriented programming. In *Proceedings of ECOOP'97*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.
[12] R. Hayton and ANSA Team. FlexiNet Architecture. ANSA Architecture Report, Citrix Systems Ltd., Cambridge, UK, February 1999. http://www.ansa.co.uk.
[13] JacORB Team. *JacORB 2.1 Programming Guide*, 2004. http://www.jacorb.org.
[14] Object Management Group. *CORBA Components, Version 3.0*, Jun 2002. OMG document formal/02-06-65.
[15] OpenCCM web site, 2003. http://www.objectweb.org/openccm/.
[16] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. Wiley, 2000.
[17] Sun Microsystems. *Java Management Extensions — Instrumentation and Agent Specification, v1.1*, 2002.
[18] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1*, 2003.
[19] Sun Microsystems. *Java 2 Platform Enterprise Edition Spec., v1.4*, 2003.
[20] S. Vinoski. Toward integration: Chain of responsibility. *IEEE Internet Computing*, 6(6):80–83, 2002.