The JBoss Extensible Server

Marc Fleury
The JBoss Group, LLC
► Francisco Reverbel
CS Department, University of São Paulo

Outline

- Introduction
- JMX Foundation
- Service Components
- Meta-Level Architecture for Generalized EJBs
- Ongoing and Future Work
- Related Work
- Concluding Remarks

Introduction

Application Servers

- Middleware platforms for development and deployment of component-based software
- Application component models
  - J2EE (Servlets/JSP and EJB), .NET, CCM
- Component-based techniques should not be applied on user applications only
  - Component-based application servers
  - Dynamically deployable middleware components
  - Two kinds of components
    - Middleware components and application components

JBoss

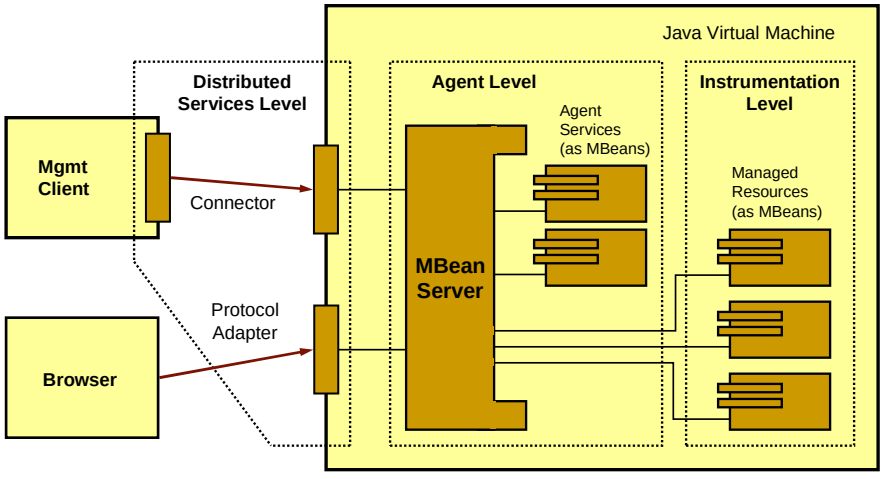
- Extensible, reflective, and dynamically reconfigurable Java application server
- At the confluence of research areas such as
  - Component-based software development
  - Reflective middleware
  - Aspect-oriented programming
- Open-source and free (LGPL)
  - 2M downloads in 2002, 1.5M so far in 2003
- Includes a set of components that implement the J2EE specification

JBoss: Open-Ended Middleware

- Users can extend middleware services by dynamically deploying new components into a running server
  - The foundation
    - Java Management Extensions (JMX) specification
    - Architecture for dynamic management of application, system or network resources
  - The building
    - Service components
      - A JMX-based model for middleware components
    - The EJB subsystem
      - Invokers, containers, dynamic proxies, and interceptors

JMX Foundation

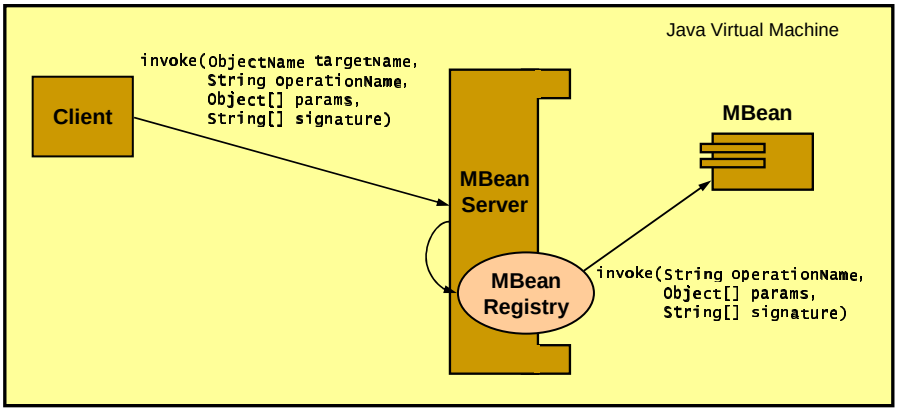
JMX Architecture



The JMX Component Model

- The instrumentation and agent levels define an in-process component model
- The MBean server provides a registry for JMX components (MBeans)
- Each MBean is assigned an object name that is unique in the context of the MBean server
- Clients use object names to refer to MBeans
- MBean operation invocations always go through the MBean server

Method invocation on a dynamic MBean



- The client holds no direct Java references to the MBean
- It does not need to know the MBean's Java class/interfaces
- This very simple arrangement favors adaptation

Dynamic and Standard MBeans

- JMX defines two kinds of MBeans
  - A dynamic MBean implements a predefined Java interface and relies on metadata to specify its management interface
  - A standard MBean 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

Dynamic MBeans

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

- The metadata class MBeanInfo supports MBean introspection
- Management attributes and operations does not need to correspond to Java fields and methods

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

```
class Foo implements FooMBean {
  ...
}

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

- The FooMBean interface follows JavaBean-like rules to represent management attributes and operations

MBean Server 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 operationName,
                Object[] params, String[] signature);
  MBeanInfo getMBeanInfo(ObjectName name);
}
```

Service Components

Issues not covered by JMX

- Dependencies between MBeans
- Service lifecycle
- Packaging and deployment of MBeans

JBoss addresses these issues with the notions of service MBean and deployable MBean

The JBoss Component Model

- Service MBeans:
  - MBeans whose management interfaces include service lifecycle operations
  - Also called service components
- Deployable MBeans:
  - Service MBeans packaged according to EJB-like conventions, in deployment units called service archives (SARs)
    - A SAR includes a service descriptor, an XML file that conveys information needed at deployment time
  - Also called deployable services
    - They are a JBoss specific extension to JMX

Service Lifecycle

- A service component may be in the stopped state or in the started state
- At each state transition one of the following lifecycle operations is invoked on the MBean:
  - `create`
  - `start`
  - `stop`
  - `destroy`

A Service Descriptor File (1 of 3)

```
<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>

   ...
```


A Service Descriptor File (2 of 3)

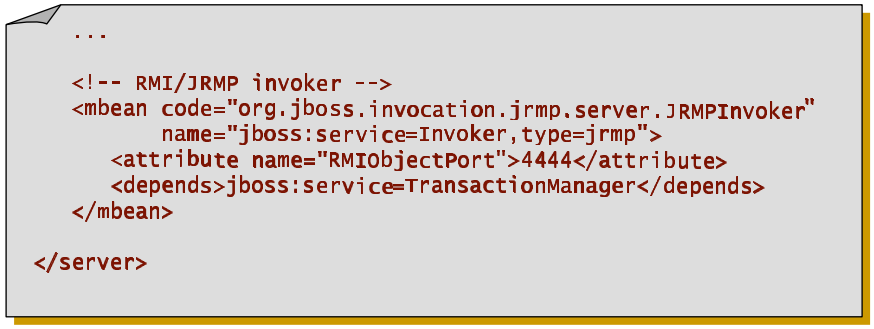
```
   ...

   <!-- 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>
      <depends>jboss:service=TransactionManager</depends>
      <depends>jboss:service=WebService</depends>
   </mbean>

   ...
```

A Service Descriptor File (3 of 3)



```
    ...

    <!-- 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>
```

- Note the “depends” elements
- JBoss manages dependencies between deployable MBeans

Dependency Management

- JBoss employs a variant of the component configurator pattern to control the lifecycle of deployable services
  - Deployment of SAR files with service MBeans are handled by a `SARDeployer`
  - The `SARDeployer` plays the role of component configurator
  - A `ServiceController` plays the role of component repository
- Deployment/undeployment events drive the lifecycle of deployable services

Deployment and Undeployment

- A `MainDeployer` handles all deployment units (SARs, JARs, EJB-JARs, WARs, RARs, etc.) by delegating the actual deployment tasks to sub-deployers:
  - `SARDeployer`, `JARDeployer`, `EJBDeployer`, …
  - The set of sub-deployers is open-ended
  - Sub-deployers are service MBeans
    - They register themselves with the `MainDeployer`, which is also a service MBean
    - `MainDeployer`, `JARDeployer`, and `SARDeployer` are not deployable components
    - All other deployers are deployable MBeans

Management interface of the MainDeployer

```
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();
  ...
}
```

Hot Deployment

- Just drop deployment units into a well-known directory
- A `DeploymentScanner` monitors the files in this directory
  - The `DeployamentScanner` 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

Class visibility is a function of time!

Class Loading Issues

- A number of application servers use variants of a class loading approach that could be called loader-per-deployment
  - This approach creates a namespace per deployment unit
  - It hinders local interactions between separately deployed parts
  - Acceptable for application components
  - Unsuitable for the dynamically deployed parts of an extensible system such as JBoss

What is bad about the Parent Delegation Model?

- Middleware components need to share non-system classes in order to interact within a VM
- So they must be loaded by a set of class loaders with a common ancestor, which loads the classes they share
- This leads to a hierarchical deployment process (a “deployment tree”)
  - Cumbersome in dynamic environments
  - Does not match the DAG nature of component dependencies

Unified Class Loaders

- A collection of unified class loaders acts as a single class loader
- It places in a single (flat) namespace all classes it loads
  - This is a significant departure from the hierarchical class loading model introduced in JDK 1.2
- Instances of `UnifiedClassLoader` are registered with a `UnifiedLoaderRepository`
  - They behave as a single `URLClassLoader` that allows its collection of URLs to be updated at any time

Dynamic Proxy Usage

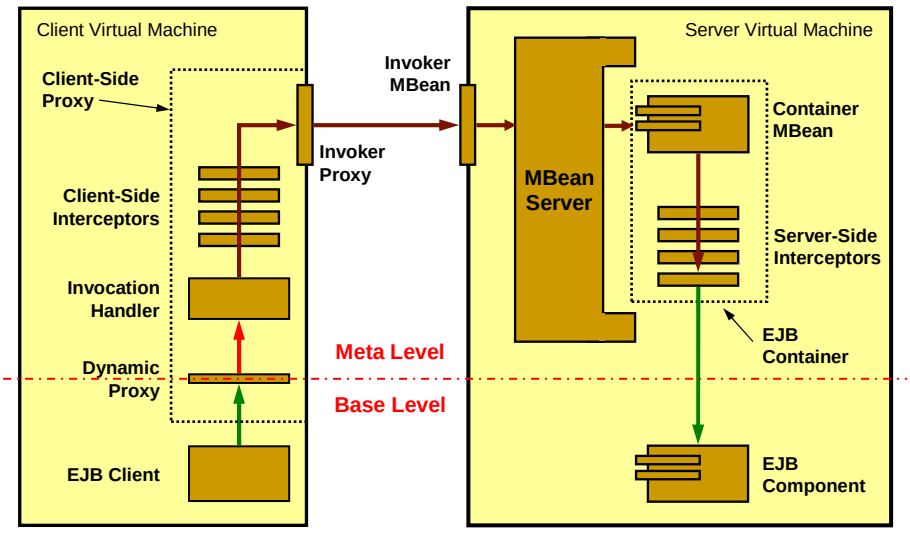
- A dynamic proxy is an object adapter that converts the type-independent interface of its invocation handler into a list of interfaces specified at runtime
- Dynamic proxies bridge the gap between the interfaces that are application-specific and those exposed by middleware components

The Dynamic Stub Idiom

- In Java RMI:
  - Serializable types are normally passed by value
  - Remote types are normally passed by reference
  - What about a remote object that is also serializable?
    - If it has not been exported through the RMI system, then it will be passed by value (in serialized form)
    - This allows the creation of custom stubs, which interact over a custom protocol with the remote objects they represent
- JBoss uses dynamic proxies as custom stubs
  - The dynamic proxy implements application interfaces
  - Its customized part is the invocation handler

Meta-Level Architecture for Generalized EJBs

Meta-Level Architecture



Reified Method Invocations

```
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
}
```

- No interface along the reified invocation path depends on base-level application types

Invoker Architecture (1 of 3)

- A powerful and flexible remote invocation architecture
  - EJB containers expose a type-independent `invoke` operation
  - Protocol-specific invoker MBeans make this operation accessible to remote clients through various protocols (JRMP, IIOP, HTTP, SOAP)
  - Client-side stubs are dynamic proxy instances
  - They convert calls to the typed interfaces seen by clients into `invoke` calls on remote invokers

Invoker Architecture (2 of 3)

- Each client-side proxy has an 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
- Interface exposed by the JRMP invoker:

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


Invoker Architecture (3 of 3)

- A client-side proxy (or, more precisely, its invocation handler) must “know a remote invoker”
- This knowledge is protocol-specific
- It is encapsulated within an invoker proxy
- A local invoker handles the case of in-process calls in an optimized way
- IIOP is treated as a special case in JBoss
  - Reason: interoperability with clients written in other languages

Generalized EJB Containers

- A container MBean is created when an EJB is deployed
- It provides middleware services to its EJB
  (instance pooling, instance caching, persistence, security, transactions...)
- … by merely aggregating aspects that do the real work
- Container configurations (XML files)
  - For standard kinds of EJBs
  - For JBoss-specific extensions
  - Customized containers for generalized EJBs

Interceptors

- Weavable aspects
- Interceptor chains interposed at the client-side and at the server side
  - Client-side interceptors
    - Aspects that involve some form of context propagations (e.g., transactions and security)
    - Handle certain invocations that can be fully processed at the client side
  - Server-side interceptors
    - Transaction, security, logging, gathering of statistical data, entity instance locking, detection of reentrant calls, management of relationships between entities

Ongoing and Future Work

Ongoing Work and Next Steps

- EJB 2.1 compliance
- Performance optimizations
- AOP framework (JBoss 4.0)
  - Class, method, field, and constructor pointcuts can be dynamically attached to any POJO
- Extensions to the meta-object protocol supported by generalized EJB containers

Related Work

JBoss owes a lot to many systems…

- Flexinet (Hayton et al)
  - Flexible remote invocation paths
  - Dynamic stubs
- Yasmin (Deri)
  - Hot-deployable components (“droplets”)
- OpenCOM (Clarke et al)
  - Lightweight component model
  - Based on a subset of COM
  - Dependence management, reconfiguration, method call interception

Concluding Remarks

JBoss...

- Demonstrates that application servers can be built out of dynamically deployed components that provide middleware services to application components
- Brings reflective middleware to the world of mainstream computing

Thank you for your attention!

- Q & A ...

Marc Fleury
marc@jboss.org

Francisco Reverbel
reverbel@ime.usp.br

http://www.jboss.org