

Use Apache Geronimo to build a cluster, Part 2: Developing the cluster nodes

Skill Level: Intermediate

[Matthew Jording \(kingrabbit@gmail.com\)](mailto:kingrabbit@gmail.com)
Freelance writer
Sensis Corporation

06 Jun 2006

Preserving the state of the applications running between nodes requires node intercommunication. Continue exploring Apache Geronimo's support for clustering in this installment, Part 2 of this five-part series. First you'll build up the cluster nodes and test communication among other nodes and the cluster manager Web service introduced in [Part 1](#). Then you'll deploy and test the current state of the application on Geronimo.

Section 1. Before you start

This five-part tutorial series is for the Java EE application developer, application server administrator, or service-oriented architecture (SOA) provider who needs production-quality application servers to serve up Web applications, Web services, or other SOA applications. It's also for those who want to understand the esoteric art of zero-downtime customer service and for those who want to learn the solution to unavailable and poorly scalable applications.

About this series

In this five-part series, explore Java EE clustering for reduced downtime and higher performance. Using the latest version, 1.0, of the Apache Geronimo Java EE application server, clustering for free has never been easier. In this series, you'll use the Geronimo server's newly added built-in clustering support to ensure production-quality performance from your applications. You'll also learn how to

deploy to multiple machines and how to ensure load balancing.

[Part 1](#) of the series introduced you to clustering, the responsibility of the cluster nodes, and managing the nodes in an application server cluster. Because of their decoupled service-oriented nature, Web services are a natural choice for our cluster manager example. The cluster manager Web service monitors and manages each node and controls the failover and load-balancing abilities of the cluster overall. By introducing the management of a cluster, you were given a hands-on approach to the guts of application server clustering.

This installment, Part 2 of the series, shows you how to create the cluster nodes. You'll revisit the cluster manager, ensuring communication between the manager and the nodes and testing communication among the nodes themselves. Node intercommunication is essential to preserve the state of the applications distributed and running between them. Last, you'll create, deploy, and monitor a Hello Cluster application to test on the cluster.

Part 3, "Load balancing and failover," will demonstrate the essential parts that comprise Java EE clustering: load balancing and failover. *Load balancing* means that the cluster manager Web service knows how much of a load each node in the cluster is experiencing and balances the load accordingly. *Failover* is where a cluster node fails. The cluster manager needs to know when failover occurs to transfer that node's responsibilities to other nodes in the cluster. You'll test the capabilities of Apache Geronimo to handle these two schemes to bring your applications to a production-quality deployment.

In Part 4, "Starting and stopping the cluster," you'll add the ability to start a cluster based on its last saved state to the cluster manager Web service. You'll also get an introduction to working with Geronimo GBeans and to Geronimo's embedded database, Apache Derby. Derby will be the container with which you persist the state of the cluster. You'll test both an intentional and an *unclean* shutdown of a cluster to determine if the state of our cluster can be safely preserved.

Part 5 will bring it all together in "Collecting statistics, deployment, and testing on multiple machines," which will demonstrate how Geronimo performs in a true clustering situation by collecting and analyzing statistics about its performance. You'll revisit the Hello Cluster application to test the functionality of the cluster. Finally, you'll deploy and test the cluster as a whole on Geronimo, and you'll set up the cluster manager Web service and multiple nodes on multiple machines and test them.

About this tutorial

In the [first installment of this series](#), you explored Geronimo as an ideal platform for high-performance environments. You also explored the various components that

provide Java EE services to the Geronimo kernel and defined an interface for interacting with the nodes within a cluster. In this installment, you'll take a look at setting up the various nodes in the cluster and test internode communication.

System requirements

You need to install the following tools to build the cluster manager Web service:

- [Java Platform, Standard Edition \(Java SE\) 5.0 Java Development Kit Version 5](#)
- [Apache Geronimo](#)
- [Apache Axis](#)
- [Maven 2.0](#)

You also need an extra PC to set up your second test node on a local network.

Install the nodes

You need two machines with Apache Geronimo 1.0 installed on each. Use the installation procedure from [Part 1](#) to install Geronimo. The initial installation is trivial and should not take long.

Give each of your nodes names that you can recall easily. I used Node0 and Node1 as a demonstration of my creativity. I found it best to use these names as the network names (that is Node0.localdomain and Node1.localdomain).

Prepare the nodes

Geronimo 1.0 is production ready for single instances, but it needs some additional configuration and customized development for a full-service cluster to work. As we discussed in Part 1, the cluster manager needs to hook into Geronimo's various tiers to provide the statistics and analysis we would like to see. For this you'll be leveraging each tier's clustering GBeans.

Section 2. Catalina clustering with JMX

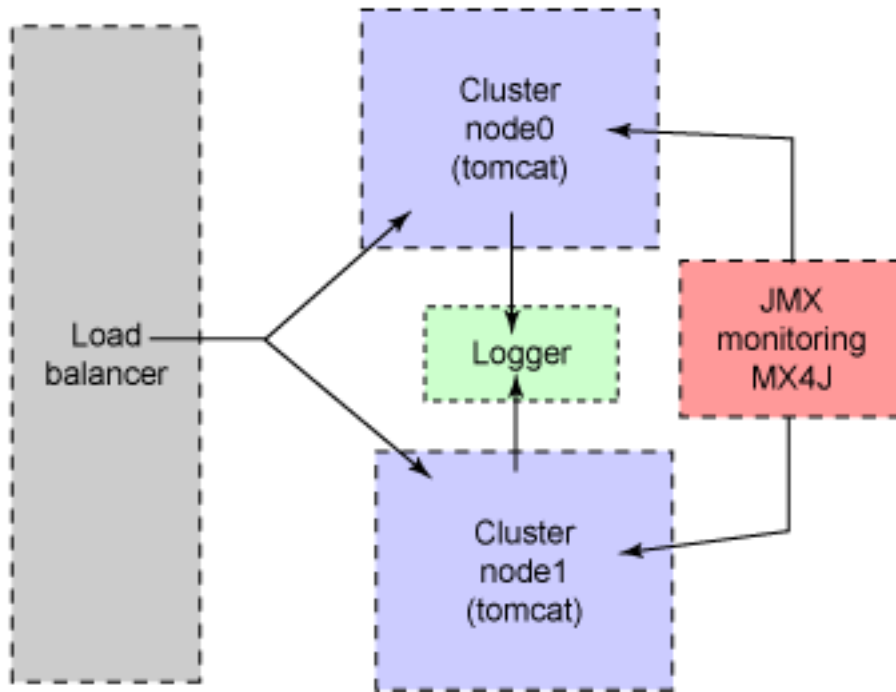
Apache Geronimo comes with either Jetty or Tomcat as its Web-tier Java EE component. Although there is some debate as to which container is the best, Tomcat is by far the most popular. A lot of work has been done for clustering Tomcat as a

stand-alone servlet container. Cluster monitoring and management for Tomcat has achieved maturity over the last few years as well. By understanding clustering and cluster management in Tomcat, you'll understand how to communicate with the Web tier in each node of a cluster.

Old-school Web-tier cluster management

Traditionally, Tomcat cluster management and monitoring has relied on the cluster supporting Java management objects (MBeans) within Tomcat itself. JMX support was available in the Tomcat 4.1 release, but has been enhanced in Tomcat 5 to provide comprehensive server monitoring and administration using Java Management Extensions (JMX). JMX is now a core part of Catalina, Tomcat's servlet implementation. Tomcat 5 uses JMX MBeans for implementing the manageability of Tomcat. You can now manage all of Tomcat's internals, such as service, engine, and host, using JMX technology (see [Figure 1](#)).

Figure 1. Tomcat and JMX



MX4J

Tomcat uses the same JMX open source framework as Geronimo -- MX4J -- to implement JMX support. You don't need to install MX4J separately to start using JMX; both Tomcat 5 and the Geronimo 1.0 installation packages include MX4J as part of the distribution.

The .jar file for the MX4J API is located in the TOMCAT_HOME/bin directory (in the file jmx.jar, which also includes the core JMX API). In Geronimo, MX4J is a module. You can see all of the deployed modules in your Geronimo 1.0 server by looking in <GERONIMO_HOME>/repository/.

MX4J is found in <GERONIMO_HOME>/repository/mx4j/jars/.

Tomcat has some base clustering built into it. So as not to reinvent the wheel, you can leverage the hard work done by Tomcat's developers as your Geronimo Web-tier clustering. The most obvious place to look for clustering in Tomcat is in the module entitled *catalina-cluster*. If you feel so inclined, download the source for Tomcat and examine the package for yourself. Let's go through some of the clustering classes within the Catalina cluster module of Tomcat:

- `DeltaManager` -- Manages replicated sessions by only replicating the deltas in data. For applications written to handle this, the `DeltaManager` is the optimal way of replicating data. The `DeltaManager` is a great way to get at the server session reporting functionality.
- `ReplicationValve` -- Implementation of a valve that logs interesting contents from the specified request (before processing) and the corresponding response (after processing).
- `SimpleTcpCluster` -- Responsible for setting up a cluster and providing callers with a valid multicast receiver/sender.

These are the base classes that Tomcat/Catalina clustering hinges upon. `DeltaManager` is used as a hook to individual node data; `ReplicationValve` is an extension of the valve interface to a cluster request; and `SimpleTCPCluster` allows Tomcat to share sessions across a multicast protocol.

Tomcat has MBeans that enable monitoring and management of a cluster. With a standard JMX browser, you can browse the clustering MBeans in a running implementation of Tomcat. If you try the same thing with Geronimo, you'll be confused, if not disappointed. The embedded Web tier talks to the Geronimo kernel, but the kernel doesn't expose the MBeans in Catalina -- for that we need to explore GBeans.

Section 3. Geronimo clustering -- GBeans

The implementation components of Geronimo don't dictate dependencies to the kernel. This is a good thing. With a lightweight independent kernel, Geronimo does

something no other Java EE-compliant server has done yet -- it becomes pluggable. If you want to use a different EJB implementation, then it only takes a deployment of your favorite EJB implementation to plug into the kernel. This decoupling is why we don't see the Catalina MBeans in a JMX explorer and why you need to deploy GBeans to use Tomcat's management and monitoring features.

Geronimo Web-tier cluster management

In [Part 1](#), Geronimo's architecture and the GBean concept were briefly discussed. Again, the concept of a GBean is significant because it assists in decoupling the architecture. This decoupling makes it lightweight and easy to reconfigure into whatever your particular business could assemble from the suite of powerful Java EE components. You'll even learn how to build upon the architecture later by creating your own GBean.

So what about all of the work that has been done on JMX? One of the coolest things about GBeans is the `MBeanServerKernelBridge`. The `MBeanServerKernelBridge` registers each GBean loaded in Geronimo as an MBean in its embedded JMX server, MX4J. So even if you've developed a mature JMX monitoring application for Tomcat, all of your work need not go to waste. Exposing MBeans with Geronimo is merely a matter of wrapping them in a GBean and registering them in the Geronimo kernel.

You'll take a look at how the Geronimo developers have done this with the valve interface for Tomcat, exposing a whole host of services from Tomcat to Geronimo's kernel. First, though, here's a description of what it takes to be a GBean.

The `FirstGBean` (see [Listing 1](#)) is a nearly bare-bones implementation. Strictly speaking, a GBean need only implement a static method that creates `GBeanInfo` that the Geronimo kernel will look for. In our example, `FirstGBean`, you'll also implement the `GBeanLifecycle` interface. The `GBeanLifecycle` interface allows the Geronimo kernel to manage the life cycle of the GBean. When it starts up, its shutdown cycle can be accessed by the kernel, which initiates each of these states.

Listing 1. FirstGBean

```
public class FirstGBean implements GbeanLifecycle{

    private static final GBeanInfo GBEAN_INFO;
    private final String objectName;
    private boolean started = false;

    static {
        GBeanInfoBuilder infoFactory = new
        GBeanInfoBuilder(EchoServer.class.getName(), EchoServer.class);
        infoFactory.addAttribute("objectName", String.class, false);
        GBEAN_INFO = infoFactory.getBeanInfo();
    }
    public void doStart throws Exception{
        // Run during instantiation of GBean.
```

```
    }  
    public void doStop throws Exception{  
        // Called before the GBeans life ends.  
    }  
    public GBeanInfo getGBeanInfo(){  
        return GBEAN_INFO;  
    }  
    static {  
        // Create the GBean Info by using the GbeanInfoBuilder  
    }  
}
```

`FirstGBean` is a first cut at a `GBean`. You can see that the static implementation for `GBean` information can be fairly straightforward. The unimplemented `doStart` and `doStop` methods are available to do any initialization and cleanup within your `GBean`. To see how to migrate from an existing class to `GBeans`, take a look at the Catalina valve interface as implemented by the Geronimo developers.

ValveGBean.java

Tomcat uses the valve interface to allow for pre- and post-process request logging, among other tasks. An individual valve can perform the following actions in the specified order:

1. Examine and/or modify the properties of the specified request and response.
2. Examine the properties of the specified request, completely generate the corresponding response, and return control to the caller.
3. Examine the properties of the specified request and response, wrap either or both of these objects to supplement their functionality, and pass them on.
4. If the corresponding response was not generated (and control was not returned), call the next valve in the pipeline (if there is one) by executing `context.invokeNext()`.
5. Examine, but not modify, the properties of the resulting response (which was created by a subsequently invoked valve or container).

You'll want to expose the valve and use it as a hook to our Web service. The best way to do this is to use the `GBean ValveGBean`. As I mentioned earlier, the `GBean` exposes a class to the Geronimo kernel management. By using the `ValveGBean`, you can hook into all of the additional valve implementations within Tomcat.

Of the Java EE components in Geronimo, the Geronimo-Tomcat module is the furthest along in `GBean` exposure. Following is an example of a `GBean`

implementation of the Tomcat valve interface. In Part 3 of this series, you'll use `ValveGBean` to access several of the Tomcat valve implementations. If in the future you wanted to implement a single sign-on service, you could use the `ValveGBean` as an interface to the `SingleSignOn` class member of the authentication package. The `SingleSignOn` class implements the valve interface, which is exposed by the `ValveBean` below. [Listing 2](#) is a full example of a Tomcat interface wrapper.

Listing 2. ValveGBean

```
package org.apache.geronimo.tomcat;
import java.util.Map;
import org.apache.catalina.Valve;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.geronimo.gbean.GBeanInfo;
import org.apache.geronimo.gbean.GBeanInfoBuilder;
import org.apache.geronimo.gbean.GBeanLifecycle;
public class ValveGBean extends BaseGBean implements GBeanLifecycle,
    ObjectRetriever {
    private static final Log log = LogFactory.getLog(ValveGBean.class);
    public static final String J2EE_TYPE = "TomcatValve";
    private final Valve valve;
    private final ValveGBean nextValve;
    private final String className;
    public ValveGBean(){
        valve = null;
        nextValve = null;
        className = null;
    }
    public ValveGBean(String className, Map initParams, ValveGBean
nextValve) throws Exception{
        if (className == null){
            throw new IllegalArgumentException("className cannot be
null.");
        }
        if (nextValve != null){
            if (!(nextValve.getInternalObject() instanceof Valve)){
                throw new IllegalArgumentException("The class given as
the NextValve attribute does not wrap an object of
org.apache.catalina.Valve type.");
            }
            this.nextValve = nextValve;
        } else {
            this.nextValve = null;
        }
        this.className = className;
        valve = (Valve)Class.forName(className).newInstance();
        setParameters(valve, initParams);
    }
    public void doStart() throws Exception {
        log.debug(className + " started.");
    }
    public void doStop() throws Exception {
        log.debug(className + " stopped.");
    }
    public void doFail() {
        log.warn(className + " failed.");
    }
    public Object getInternalObject() {
        return valve;
    }
    public ValveGBean getNextValve() {
        return nextValve;
    }
    public static final GBeanInfo GBEAN_INFO;
```

```
static {
    GBeanInfoBuilder infoFactory =
GBeanInfoBuilder.createStatic(ValveGBean.class, J2EE_TYPE);
    infoFactory.addAttribute("className", String.class, true);
    infoFactory.addAttribute("initParams", Map.class, true);
    infoFactory.addReference("NextValve", ValveGBean.class,
J2EE_TYPE);
    infoFactory.addOperation("getInternalObject");
    infoFactory.addOperation("getNextValve");
    infoFactory.setConstructor(new String[] { "className",
"initParams", "NextValve" });
    GBEAN_INFO = infoFactory.getBeanInfo();
}
public static GBeanInfo getGBeanInfo() {
    return GBEAN_INFO;
}
}
```

One of the most important parts of this class is the implementation of `ObjectRetriever`, which is meant to return objects that are internal to the Geronimo modules. These objects need not be registered with the kernel to be used by other modules.

Section 4. Your first GBean

With a base understanding of how the kernel manages GBeans and a complex example of a GBean implementation, you're now ready to begin programming GBeans. GBeans and the modules that group them together are the building blocks for extending the functionality of Geronimo.

The GBean, POJO, and JMX connections

Tomcat MBeans are mostly invisible to Geronimo, except for the few exposed by GBeans, which were added to give Geronimo-Tomcat the functionality normally extended to applications via JMX. So to exercise a clustering example and make use of the component, you first need to expose Tomcat's clustering functionality.

A GBean is a simple Plain Old Java Object (POJO), except that it must plug into the `GbeanInstance` by implementing `GbeanLifecycle`. By doing so, Geronimo can manage the bean and expose the methods to other services and modules.

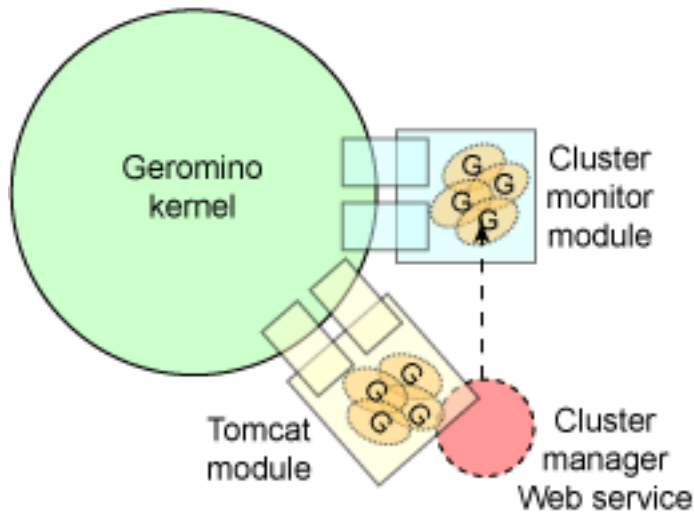
Cluster monitoring

Geronimo is a lightweight container that is -- as a matter of necessity -- ignorant of

its Java EE implementing components. Tomcat has clustering built into its components and has been expanded upon to report to the Geronimo kernel through the use of GBeans; but what a full Java EE node needs is a suite of statistical reporting that can coordinate between all of the enterprise components. You'll eventually need a data aggregation engine. Because the kernel doesn't need to know all of the implementation details of its various components, what seems to be the obvious node switchbox point is unavailable to use. For nodes to report their status properly to the cluster manager Web service, you must create GBeans that hook into the Tomcat management. These GBeans fill the same role as the JMX MBeans you explored earlier.

Within the current configuration, you'll create your own Geronimo module, which will extend the Tomcat GBeans used by the cluster manager Web service.

Figure 2. Create GBeans based on common MBeans used within Tomcat cluster configurations



You create the following GBeans, which are based on common MBeans and used within Tomcat cluster configurations. You'll use many of these beans later in this series. Below is a brief description of the classes:

- `NodeUtility` -- The utility class for defining the cluster GBeans instantiation and registration in the Geronimo kernel. The custom Axis context listener class calls the methods in this class to register GBeans when node Web service context is initialized.
- `NodeMonitor` -- Has methods to get GBean attribute details for the manageable server components. The cluster manager console application calls these methods to get the details of node cluster components.
- `CustomDeltaManagerGBean` -- The custom session manager class

created to access the session information. It provides the getter methods for session attributes to get the details of an HTTP session.

- `CustomReplicationValveGBean` -- The custom replication valve created to access the request filter details. It was created so the filter pattern (*.js, *.jpg, *.html, and so on) can be viewed and modified dynamically using GBean methods.
- `ClusterContextListener` -- The custom Axis context listener created to call the GBean instantiation methods when the Geronimo server is started.
- `WebServiceLogger` -- Implementation of Log4J for logging debug messages in the Web service.

Take special note of the `CustomDeltaManagerGBean`, as that is the class that provides the Web service with the session information about nodes it may inspect.

Section 5. Deploy your Hello Cluster module

As you learned in [Part 1](#) of this series, deploying an application is easy, and so is deploying a group of GBeans. Every component within Apache Geronimo, including Tomcat, is merely a group of GBeans. By understanding how to package and deploy a group of GBeans, the key to extending Geronimo's functionality is yours.

Geronimo configurations and modules

After creating the GBeans, you must make them available to any component that needs to access them. You do this by packaging the GBean in configuration. The packaging is straightforward, and you can use the standard Maven package declaration to create a JAR. The easiest way to register your module is to copy the JAR to its own directory structure in the Geronimo repository directory structure. In our case, the directory structure would be

```
<GERONIMO_HOME>/repository/ClusterMonitor/jars/.
```

Copying the packaged JAR of our module's classes in this directory allows the kernel to find and register the module, given its deployment plan.

Module deployment plan

The configuration file for a module is a standard XML file specifying all the various

class dependencies, variable values, and kernel parameters the GBean might need. The deployment plan is based on a Spring configuration, like Inversion of Control (IoC). The module configuration is a simple XML file with a standard document tree. Configuration is the root element of a module deployment with the following elements as its children:

- `configId` -- The identifier for the module.
- `parentId` -- An optional child element that defines a parent module of which the same access rules to subclasses apply.
- `domain` and `server` -- Either a domain or server must be defined, or you must have the `parentId` value defined.
- `InverseClassloading` -- If this Boolean is set to `true`, the parent module's classes are overridden by this, the child.
- `Import` -- An import of another module. These imports are added to the current configuration. You can add these modules in the same syntax as a Maven import, shown in [Listing 3](#). The module imported in [Listing 3](#) is referenced by the URI `ClusterMonitor/service/1.0/war` in a Maven-style import statement.

Listing 3. Maven import syntax

```
<import>
  <groupId>ClusterMonitor</groupId>
  <artifactId>service</artifactId>
  <version>1.0</version>
  <type>war</type>
</import>
```

- `include` -- Acts more like a regular Maven dependency, physically copying the archive specified in the configuration from the Geronimo repository to your module.
- `dependency` -- A repository reference, adding the referred library to the class path of your module.
- `Hidden-classes` -- A CSV formatted list of classes that should not be loaded from the parent modules.
- `Non-overrideable classes` -- Specifies a list of classes that are final and cannot be overridden by children.
- `gbean` -- Defines the custom service itself.
- `Attributes` -- Specifies the attribute name, type, and values.
- `Xml-attributes` -- Allows for attribute specification via

xml-attribute syntax.

- `Reference` -- A reference name, type, and reference pattern by which to identify a GBean.
- `References` -- Allows you to configure multipatterned references.
- `Xml-reference` -- Specifies XML reference values as an XML string.
- `dependency` -- Specifies the dependencies of this GBean to other GBeans.

[Listing 4](#) is the deployment plan for our `ClusterMonitor` configuration or module. Every module in Geronimo has a plan. For additional examples, you may want to look at the source of the modules under a plan subdirectory.

Listing 4. clusterManagerPlan.xml configuration plan

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://geronimo.apache.org/xml/ns/deployment"
configId="sample/clusterMonitor">
  <dependency>
    <uri>myservices/ClusterMonitor/1.0/jar</uri>
  </dependency>
  <gbean name="geronimo.apps:name=com.ibm.clustermonitor.CustomDeltaManagerGBean"
class="ClusterMonitor">
    <attribute name="port" type="int">4545</attribute>
  </gbean>
</configuration>
```

In the configuration plan shown in [Listing 4](#), you can see that you declare the GBeans you use. Each is listed to provide the GBean hooks that Geronimo uses. Additionally, the actual JAR we deposited earlier in the repository path as a dependency is shown.

Again, the cluster monitor Geronimo module is in a Maven 2 file set that can be packaged with the `mvn` package command. Once packaging is done and your JAR is in the repository, the module descriptor can deploy and register the module with the following Geronimo deployer:

```
%java -jar bin/deployer.jar deploy clusterMonitorPlan.xml
```

ClusterManagerSoapBindingImpl

Now that you have a module deployed, you can access the GBeans in your Web service. Because you defined the interface for `ClusterManagerService` and generated the implementation stubs, all you need to do is interface with the GBeans

for clusters (see [Listing 5](#)).

Listing 5. ClusterManagerSoapBindingImpl.java

```
package com.ibm.ClusterManager;

public class ClusterManagerSoapBindingImpl implements
com.ibm.ClusterMannager.ClusterManager{
    private CustomDeltaManagerGBean deltaManager;
    public boolean stopNode(int in0) throws java.rmi.RemoteException {
        return false;
    }

    public int getNodeSessions(int in0) throws java.rmi.RemoteException
    {
        return deltaManager.getSessions();
    }

    public java.util.Calendar getNodeUptime(int in0) throws
java.rmi.RemoteException {
        return null;
    }

    public boolean startNode(int in0) throws java.rmi.RemoteException {
        return false;
    }

    public java.lang.Object[] getNodeFailureReport(int in0) throws
java.rmi.RemoteException {
        return null;
    }

    public int getNodeLoadCapacity(int in0) throws
java.rmi.RemoteException {
        return -3;
    }

    public int setNodeLoadCapacity(int in0) throws
java.rmi.RemoteException {
        return -3;
    }
}
```

To implement the session monitoring of nodes, you can use the `.getSessions()` method of the `CustomDeltaManagerGBean`.

Geronimo-web.xml cluster manager deployment plan

To take advantage of GBeans and Tomcat clustering classes, you need to add a deployment descriptor to the Web service. The Geronimo-web.xml deployment descriptor is the deployment plan specific to Geronimo and separate from the generic web.xml Java Enterprise Web application deployment descriptor (see [Listing 6](#)).

Listing 6. ClusterManager deployment plan

```

<cluster>TomcatCluster</cluster>

<!-- Cluster -->
<gbean name="TomcatCluster"
class="org.apache.geronimo.tomcat.cluster.CatalinaClusterGBean">
  <attribute name="className">
org.apache.catalina.cluster.tcp.SimpleTcpCluster
</attribute>
  <attribute name="initParams">

managerClassName=org.apache.catalina.cluster.session.DeltaManager
  expireSessionsOnShutdown=false
  useDirtyFlag=false
  notifyListenersOnReplication=true
</attribute>

  <reference name="Membership">
<moduleType>J2EEModule</moduleType>
<name>TomcatMembership</name> </reference>
  <reference name="Receiver">
<moduleType>J2EEModule</moduleType>
<name>TomcatReceiver</name> </reference>
  <reference name="Sender">
<moduleType>J2EEModule</moduleType>
<name>TomcatSender</name> </reference>
  <reference name="TomcatValveChain">
<moduleType>J2EEModule</moduleType>
<name>ReplicationValve</name> </reference>

</gbean>

<!-- Membership -->

<gbean name="TomcatMembership"
class="org.apache.geronimo.tomcat.cluster.MembershipServiceGBean">
  <attribute
name="className">org.apache.catalina.cluster.mcast.McastService</
attribute>
  <attribute name="initParams">
mcastAddr=228.0.0.4
mcastBindAddress=xx.yy.zz.aa
mcastPort=45564
mcastFrequency=500
mcastDropTime=3000
  </attribute>
</gbean>

<!-- Receiver -->

<gbean name="TomcatReceiver"
class="org.apache.geronimo.tomcat.cluster.ReceiverGBean">
  <attribute
name="className">org.apache.catalina.cluster.tcp.ReplicationListener
</attribute>
  <attribute name="initParams">
tcpListenAddress=xx.yy.zz.aa
tcpListenPort=4001
tcpSelectorTimeout=100
tcpThreadCount=6
  </attribute>
</gbean>

<!-- Sender -->

<gbean name="TomcatSender"
class="org.apache.geronimo.tomcat.cluster.SenderGBean">
  <attribute
name="className">org.apache.catalina.cluster.tcp.ReplicationTransmit

```

```
ter</attribute>
  <attribute name="initParams">
    replicationMode=pooled
    ackTimeout=15000
  </attribute>
</gbean>

<!-- Valves -->
<gbean name="ReplicationValve"
  class="org.apache.geronimo.tomcat.ValveGBean">
  <attribute name="className">
org.apache.catalina.cluster.tcp.ReplicationValve
  </attribute>
  <attribute name="initParams">

filter=.*\.\gif;.*\.\js;.*\.\css;.*\.\png;.*\.\jpeg;.*\.\jpg;.*\.\htm;.*\.\html;
.*\.\txt;
  </attribute>
</gbean>
```

As you can see from the GBean declarations in [Listing 6](#), the Web service will have access to the Geronimo GBeans when you redeploy the service in conjunction with the new plan. Plans are essentially the glue between the various modules within Geronimo. You may have noticed that the deployment tool of the Geronimo admin console has a field for plans. When you deploy your Geronimo application, you merely have to specify its plan to attach it to the beans registered to the kernel. After redeploying the Web service, you can begin using the service by creating the cluster manager console, our simple Web service client.

Section 6. The cluster manager console

Our cluster manager needs a user interface or view in the model view controller concept. The first user interface many programmers learn to code is the console or command-line client. You'll create a quick console client to the cluster manager to demonstrate the ease of interfacing with a Web service.

The Web Service view

Next we need to create a client to the Web service interface we defined in [Part 1](#). Web services have the most flexible view separation of all enterprise applications, so we can whip up a simple console application quickly. Later you can revisit this and create an Ajax, or rich client, to act as your management console. It would be useful to have client capability in the Apache Geronimo management console as well. Creating a simple Axis command-line client is easy with Axis. You declare your endpoints and the methods to call as well as their returns. When properly wrapped in a command-line interpreter, a quick and easy way to check on your cluster nodes is

available (see [Listing 7](#)).

Listing 7. CMConsoleClient

```
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import javax.xml.namespace.QName;

public class CMConsoleClient {
    public static void main(String [] args) {
        String node = args[0];
        try {
            String endpoint =
                "http://" + node + ":8080/ClusterManagerService/services/ClusterManagerService";

            Service service = new Service();
            Call call = (Call)
                service.createCall();
            QName nodeUptime = new
                QName("getNodeUptime");
            QName nodeSessions = new
                QName("getNodeSessions");
            call.setTargetEndpointAddress( new
                java.net.URL(endpoint) );
            call.setOperationName(nodeSessions);
            ...
        }
    }
}
```

You can see in [Listing 7](#) that you've set up your `CMConsoleClient` to check on a single node's session load. The endpoint is set to listen to `node0`. Obviously, you want to have all of the nodes reporting to your client. In Part 3, you'll explore how to make the client dynamically discover the nodes in a cluster. For this tutorial, you'll keep the endpoint as a parameter for the client.

Redeploy the cluster manager

Now that you have the cluster monitoring configuration in place in Geronimo, you need to redeploy your cluster manager Web service. In Part 1, we used the management console for Geronimo, but you should know about the deployer application it relies on and the other ways to deploy and undeploy. The deploy JAR was called when we deployed the `ClusterMonitor` deployment plan -- it's that deploy JAR that can be used for all deployments (and it's preferred for its immediate error reporting):

```
Java -jar deployer.jar [options] command [command options]
```

Available commands are:

- `deploy`

- undeploy
- distribute
- start
- stop
- list-modules
- list-targets

For additional information on each of these commands, you can run the help option, as follows:

```
Java -jar deployer.jar help [command]
```

The deployer help option gives an explanation of all the options for the specified command.

The Geronimo deployer and the deploy script, `deploy.bat`, are a useful way to quickly deploy modules, applications, and their associated plans. Becoming familiar with the deploy tools in Geronimo can make frequent updates to application deployments easier.

Section 7. Communication testing

You need to ensure that the data to be conveyed from each node can be sent and received. Two tests will be run -- one to make sure that the nodes can communicate with each other and the other to test that the same information can be digested by our cluster manager console.

Inter-node communication testing

Now that the nodes are prepared, you must make sure that each node can perform its duties. In the first test, you can ensure that the node expresses its session load to its clients through the cluster manager Web service. After you get the applications installed on each node, you can test session replication by hitting the Web service with any Web service client:

```
http://node0:8080/ClusterManagerService/services/ClusterManagerService?\
```

method=getNodeSessions&node=0

Axis exposes a good deal of its methods directly within a Web interface. Because it is a Simple Object Access Protocol (SOAP) wrapper around servlet methods, we can call the above URL, and Apache Geronimo+Axis will return a complete SOAP response in a comfortable XML format (see [Listing 8](#)).

Listing 8. Response

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <getNodeSessionsResponse
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <getNodeSessionsReturn href="#id0"/>
    </getNodeSessionsResponse>
    <multiRef id="id0" soapenc:root="0"
      soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      xsi:type="xsd:int"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">3</multiRef>
  </soapenv:Body>
</soapenv:Envelope>
```

Note that the output page contains the ID of the server that is servicing the request. In your browser window, fill in the appropriate input fields and press the **Submit** button. The console dialog (the prompt where you started Geronimo) should show that this HTTP session data is being transmitted and received between the cluster members.

Now check Node1 with the same URL and watch the resultant output (see [Listing 9](#)).

Listing 9. Listing Node1 output

```
<multiRef id="id0" soapenc:root="0"
  soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xsi:type="xsd:int"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">8</multiRef>
</soapenv:Body>
</soapenv:Envelope>
```

The response difference is within the `multiRef` tag, where you'll see the full `int` response to the request. For Node0, notice there are three sessions running, and for Node1 there are eight sessions currently running.

Cluster manager console communication testing

After listing your nodes in a simple property file, you can start up the console on the

command line, passing in a parameter to list all the sessions running on each node. Pass in the method and the node to check as a parameter to the `CMClient`.

```
%java -jar CMClient.jar node0 getSessions
Session load for Node0 = 12
```

By passing in the node name and the method to execute on the node, you can retrieve all the needed information from the cluster nodes.

Section 8. Prepare for load balancing

The next installment in this series will show different ways to load balance the current cluster. To prepare for load balancing, you need to rework the deployment of the nodes you tier.

Each Apache Geronimo cluster member must have a unique `jvmRoute` designation. The `jvmRoute` attribute allows the mod-jk load balancer to provide a *sticky session* (sending all requests for the same HTTP session to the same cluster member). This is possible since the load balancer places the `jvmRoute` value in the session cookie (or encoded URL) that is returned to the Web browser.

Set the `jvmRoute` attribute

You can set the `jvmRoute` attribute by updating the `var/config/config.xml` file by adding the lines indicated in bold in [Listing 10](#).

Listing 10. `var/config/config.xml`

```
<configuration name="geronimo/tomcat/1.0/car">
  <gbean name="TomcatResources">
  </gbean>
  <gbean name="TomcatEngine">
    <attribute name="initParams">
      name=Geronimo
      jvmRoute=nodeX
    </attribute>
  </gbean>
  <gbean name="TomcatWebConnector">
    <attribute name="host">0.0.0.0</attribute>
    <attribute name="port">8080</attribute>
    <attribute name="redirectPort">8443</attribute>
  </gbean>
```

You can see how this configuration change to the Tomcat module sets a simple

attribute of `jvmRoute` to each individual node name: `node0`, `node1`, `node2`, and so on. The configuration file is an important point of change to the module and for the greater Geronimo server. By allowing trivial changes to the configuration file to change a module, the cost of changing code is drastically reduced.

Section 9. Summary

In this installment of the series, you set up two nodes for your cluster and tested the communication between them. You saw how Tomcat, as a Web container, has moved toward clustering and how to leverage the existing clustering facilities of a Java EE tier to enhance Geronimo. You now know a great deal more about extending and customizing Geronimo to serve your needs.

In the next installment, you'll learn how to set up Apache HTTPD as a load balancer for the cluster and, alternatively, use a `LoadBalancer` module for Geronimo. You'll also see how failover works. Soon you'll appreciate how Geronimo's clustering capabilities help keep your applications open for business around the clock.

Downloads

Description	Name	Size	Download method
Cluster Manager example source code	ClusterMonitor.tar	100KB	HTTP

[Information about download methods](#)

Resources

Learn

- Get organized and consolidated [documentation for Apache Geronimo](#).
- Read "[Dependency injection in Apache Geronimo, Part 1: A new way to look at decoupling in J2EE applications](#)" (developerWorks, February 2006) and "[Dependency injection in Apache Geronimo, Part 2: The next generation](#)" (developerWorks, February 2006).
- Read this tutorial on [Distributed Multitiered Applications in Java EE](#) from Sun Microsystems.
- Get all the latest Geronimo news in [The Geronimo renegade](#) column from developerWorks.
- Check out this classic tutorial on the basics of clustering, "[Clustering: a basic 101 tutorial](#)" (developerWorks, April 2002).
- Visit the [CodeHaus site](#) for an open source project repository with a strong emphasis on the Java platform, focused on quality components that meet real-world needs.
- Visit the [Geronimo Wiki](#).
- Peruse the article, "[Migrating Apache Axis Apps to Axis2 with Geronimo](#)" (developerWorks, March 2006).
- Check out more great pieces on Geronimo:
 - "[Building a better J2EE server, the open source way](#)" (developerWorks, May 2005)
 - "[Geronimo! Part 1: The J2EE 1.4 engine that could](#)" (developerWorks, May 2005)
 - "[Geronimo! Part 2: Tame this J2EE 1.4 bronco](#)" (developerWorks, May 2005)
 - "[Three ways to connect a database to a Geronimo application server](#)" (developerWorks, June 2005)
 - "[Create, deploy, and debug Apache Geronimo applications](#)" (developerWorks, May 2005)
 - "[Apache Geronimo uncovered](#)" (developerWorks, August 2005)
- Visit the [developerWorks Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM® products.

- Check out the developerWorks [Apache Geronimo project area](#) for articles, tutorials, and other resources to help you get started developing with Geronimo today.
- Check out the [IBM Support for Apache Geronimo](#) offering, which lets you develop Geronimo applications backed by world-class IBM support.
- Find helpful resources for beginners and experienced users at the [Get started now with Apache Geronimo](#) section of developerWorks.
- Browse all the [Apache articles](#) and [free Apache tutorials](#) available in the developerWorks Open source zone.
- Browse for books on these and other technical topics at the [Safari bookstore](#).

Get products and technologies

- Get [WADI](#) (a geeky recursive acronym that stands for WADI Application Distribution Infrastructure). WADI is built on top of the ActiveCluster API to provide clustering for HttpSession.
- Get [ActiveCluster](#), a generic clustering API used by ActiveMQ and WADI. It simplifies operations that are typical for clustering applications.
- Get [ActiveIO](#), which is built on top of ActiveCluster and is used by ActiveMQ for clustering across a network of brokers.
- Check out [ActiveSpace](#), which is used by ActiveMQ to support Staged Event-Driven Architecture (SEDA)-style load-balancing data across consumers.
- Download and learn more about [MX4J](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.
- Download [Apache Geronimo, Version 1.0](#).
- Download your free copy of [IBM WebSphere® Application Server Community Edition V1.0](#) -- a lightweight J2EE application server built on Apache Geronimo open source technology that is designed to help you accelerate your development and deployment efforts.

Discuss

- [Participate in the discussion forum for this content](#).
- Stay up to date on Geronimo developments at the [Apache Geronimo blog](#).
- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the author

Matthew Jording

Matthew Jording has spent the last decade developing Java Enterprise applications for Fortune 500 companies. Currently employed by Sensis Corporation, Matthew is working on a Geronimo-based service-oriented architecture for the aerospace industry. Matthew and co-author Nathan Mittler are writing *Geronimo In Action* for Manning Publications. Matthew is married with two teenage daughters and lives in Syracuse, New York.