IBM WebSphere Developer Technical Journal: Maintain continuous availability while updating WebSphere Application Server enterprise applications

Peter Sickel

December 01, 2004

This article describes a method for rolling out a new version of an enterprise application into a production environment where continuous availability of the application is desired. Applications with browser-based clients and Java™-based clients are discussed.

Introduction

This article describes the details of moving a new version of a J2EE™ enterprise application into production, in an IBM WebSphere® Application Server Network Deployment Version 5.x cell, while maintaining continuous availability of the site. Steps are described for performing the application update using two different approaches, one using multiple WebSphere cells and another using a single WebSphere cell. Each approach has its own pros and cons, and it often makes sense to combine the two approaches in production settings, so understanding both approaches will be useful. Some brief background information on WebSphere runtime architecture and the structure of the HTTP plug-in configuration file will set the stage for the migration process.

This article assumes familiarity with IBM WebSphere Application Server V5.x and J2EE enterprise applications.

For browser-based clients, the procedure takes advantage of the ability to manipulate the routing table of the HTTP Server plug-in by modifying the `plugin-cfg.xml` file. The session affinity feature of the plug-in helps to keep existing users routed to application servers of the current version while new users are routed to application servers running the new version of the application. The LoadBalanceWeight attribute for the server element of the `plugin-cfg.xml` file is used to gracefully "drain" servers that are to be taken offline as the new version is brought online

When a migration from one version of an application to another is under consideration, there are several issues that can arise depending on the nature of the changes between application versions. Major considerations include:

- Database schema compatibility
- Work-in-process migration

- User experience considerations
- EJB version compatibility.

These considerations will likely have a larger impact on version migration planning and implementation than the mechanics of rolling the application onto the production servers. These issues are outside the scope of this article and it is assumed they are handled properly through backward compatibility, or through migration steps independent of the application rollout.

For the sake of simplicity, we will assume that the application servers are running only the application to be migrated. Realistically, application servers tend to run more than a single application. Imagine a situation where a cluster of servers is running applications A, B, and C. The recommended approach is to migrate only one application at a time, say, application A. Of course, the testing phase of development includes a set of tests to ensure the new version of application A (and all its supporting libraries) is compatible with the current version of applications B and C (and their supporting libraries). If this in fact is not possible, then the migration will need to involve a separate application server cluster to run the new version of application A. This is likely to indicate the need for additional hardware.

If the production environment is configured to use persistent HTTP session state, then there are (hopefully, low probability) failure scenario circumstances where an application server running version N of an application may access the HTTP session state of version N+1 of the application and vice versa. Suppose there is a failure of a server running version N of the application and the subsequent requests happen to be routed to an application server running version N+1 of the application. If the class definition of an object held in HTTP session state changes from one version of the application to the next, then a `java.io.InvalidClassException` will be thrown when the session state is deserialized. The application should be programmed to do something reasonable should this exception arise. The best thing to do in this circumstance is probably to force the user to a login, or otherwise redirect the user to a starting page URL, so that a new session will be created, and thus get the user back onto a server running one version of the application or the other. On the other hand, if you have servers failing during the transition, chances are you are going to immediately rollback version N+1 of the application and restore the site to servers that are all running version N of the application.

We will assume the application rollout will happen over a relatively short period of time; for example, a couple of hours, at most. If the rollout period is longer (that is, long enough that you may have the same user visiting the site for more than one new session during the rollout), then it is likely that one of the "user experience" considerations for the migration will need to be a means of making sure users who have seen the new version of the application continue to see the new version on subsequent site visits during the rollout; if a user has used the new version of the application, it is not likely you want that user going back to the old version. That could happen during an extended transition period, because servers running both N and N+1 versions of the application will be running. There would be a need to have some mechanism to persistently keep track of which version of the application the user should be directed to, such as a user profile database. This, of course, assumes all users identify themselves at the time of a site visit (through a login, for example), and there would need to be a mechanism to direct users to servers running their version. (The details of building an application to cover this scenario are outside the scope of this article.) This extended transition scenario is fundamentally different from the fast transition scenario covered by this article. In an extended transition scenario multiple versions of an application are purposely kept running for some long period of time, usually while the user population is trained to use the new version, with some users explicitly choosing to use the previous version while others use the new one.

Background on WebSphere Application Server architecture

An important feature of WebSphere Application Server V5.x is the ability to define server clusters. Members of a server cluster serve the same applications. Each member of a cluster gets a unique clone ID assigned to it, which is appended onto the end of the session cookie. The HTTP plug-in inspects the session cookie and uses the clone ID to ensure that requests belonging to the same session get routed back to the same server; this is referred to as session affinity.

Initial requests from a new user get load balanced across the members of the cluster by a simple round robin policy. Each server has a LoadBalanceWeight attribute that can be used to statically assign a preference to particular servers in the cluster. If all members of a cluster have the same LoadBalanceWeight, then they will share the load equally. If the LoadBalanceWeight of a particular server is set to 0, then only requests already associated with a session on that server are sent to it. (That server is taken out of the pool of servers eligible for handling new sessions.) This turns out to be useful when an application version update is occurring because it ensures that a user will continue to be served by the same application version during the lifetime of his or her session because all requests are going back to the same application server. Further, this also ensures that the application servers serving the old version of the application will eventually no longer have any active sessions. (How long it takes for all active sessions to end depends on the use-case scenarios for the application.)

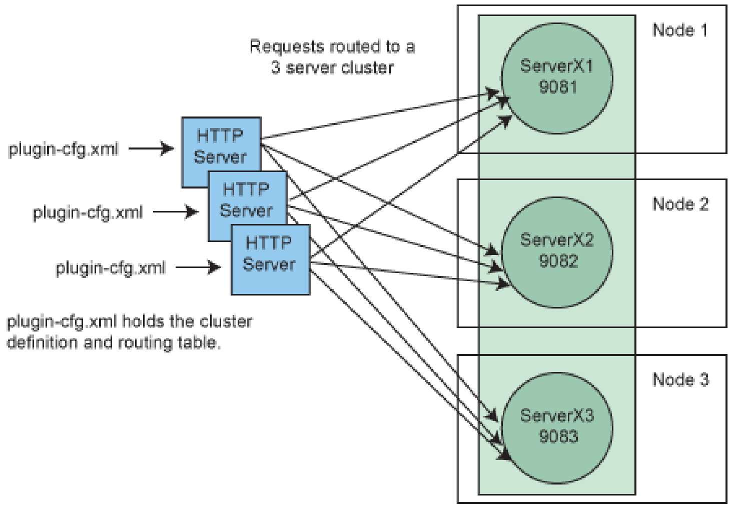
The WebSphere HTTP plug-in reads a `plugin-cfg.xml` file to determine what applications are associated with what application server clusters and builds a routing table accordingly. The `plugin-cfg.xml` file is re-read periodically by the plug-in so the HTTP server does not need to be stopped and restarted for the plug-in to discover when something in the `plugin-cfg.xml` file has changed. (The refresh interval defaults to one minute, but is configurable by an attribute defined in the `plugin-cfg.xml` file.)

Some key characteristics of clusters that are relevant to application upgrades include:

- Cluster members must all be members of the same cell.
- Applications cannot be installed on individual members of a cluster. (It is possible to turn off automatic node synchronization and use manual synchronization to enable an administrator to install an application on cluster members one at a time, but this is often outside the intended operating procedures for a cell, and these kinds of "tricks" are not recommended.)
- Application servers that are a member of a cluster are assigned a unique identifier referred to as a clone ID. The clone ID does not change and is unique to a server as long as it exists.

Figure 1 illustrates the use of a three-node cluster (named ClusterX) that handles requests from a bank of HTTP servers.

Figure 1. A server cluster defined in a WebSphere cell



To avoid diagram clutter, only one set of arrows that represent request connections from a bank of HTTP servers to a collection of application servers are shown. The numbers associated with the application servers, 9081, 9082, and 9083, represent the Web container listening port.

Rollback scenario

One of the important considerations for any rollout plan is to have a good rollback plan. Some important characteristics of a good rollback plan are:

- Uninstall/install cycle is not necessary; this has the potential for error and will likely take too long.
- Server startup is not necessary; avoid this in order to avoid server startup times.
- The rollback procedure requires a simple switching operation; for example, changing an entry in an IP sprayer routing table or `plugin-cfg.xml` file.

If your organization has a rigorous test process and your test team is afforded the time and resources to do functional and load testing in an environment with simulated load that accurately reflects production load, then the new version of the application will roll out to production without a hitch, right? Not usually. Therefore, you need a rollback plan, and the plan needs to be tested. When a new version of an application is subjected to production load, some lurking defect might be uncovered and so a rollback would be necessary. The rollout procedures described in this article satisfy these features of a rollback plan, making it quick and easy to execute a rollback.

Multiple cell approach

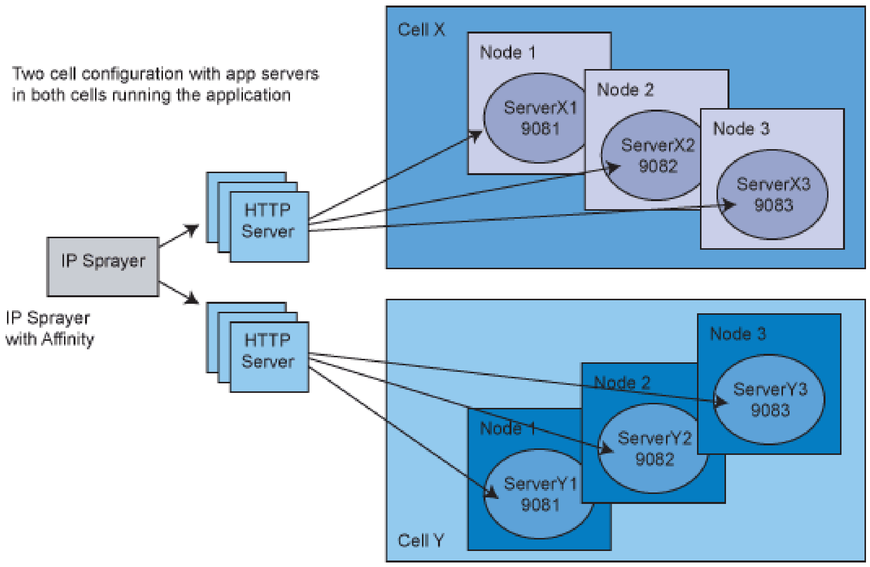
Some sites use a multi-cell configuration because it has good fault tolerance characteristics. A multi-cell configuration also makes application updates relatively simple. To simplify the explanation in this article, only two cells are used and they will be referred to as cell X and cell Y.

In a very large scale multi-cell configuration, it might be the case that there are separate banks of HTTP servers, one for each cell, with each bank of HTTP servers exclusively serving requests to just the servers in one cell. (See Figure 2.) In that case, it would be necessary to achieve session affinity at the IP sprayer. An IP sprayer (such as WebSphere Load Balancer) can do what is referred to as content based routing, and use the WebSphere Application Server session cookie with the clone ID to route:

- all requests intended for the servers in cell X to the HTTP servers associated with cell X.
- all requests intended for servers in cell Y to the HTTP servers associated with cell Y.

The IP sprayers from other vendors have similar approaches to achieving stickiness at the HTTP server tier. When you have session affinity at the HTTP server tier, the two cells operate relatively independently (with the possible exception of shared back-end servers), which leads to a very clean administrative separation of the cells.

Figure 2. Multi-cell configuration



A site of intermediate scale may have only one bank of HTTP servers, yet still have multiple cells; that one bank of HTTP servers is load balancing requests across servers in both cells. In this case, the `plugin-cfg.xml` files from each cell need to be merged to form one cluster (from the point of view of the HTTP plug-in), even though from the point of view of the WebSphere Application Server administration processes (the deployment manager for each cell) there are multiple clusters, one in each cell.

The details of an application upgrade in a multiple cell configuration will be described in the next section, but the gist of it is to take one cell out of production, install the new version of the application in that cell, begin routing requests to that cell, and then, after some burn-in period that instills confidence that the new version of the application will stand up in production, repeat the rollout process to the servers in the other cell. This procedure assumes there is sufficient capacity in the active cell to handle the load of the entire site at the time the rollout is done. (In a two cell example, taking out one cell means idling 50 percent of capacity, which may be too high depending on load conditions at the time of the rollout. The available hardware can be spread among multiple cells, such that the percentage of capacity taken off-line in one cell would be small enough to allow the remaining capacity to handle the load at the time of the rollout. The rollout procedure itself remains conceptually the same.)

Application rollout with multiple cells

This section presents steps for performing application rollout with a multiple-cell configuration. As mentioned earlier, only two cells are used here for simplicity, but the process is conceptually the same for configurations with more than two cells.

A. Starting state

The starting state is as follows:

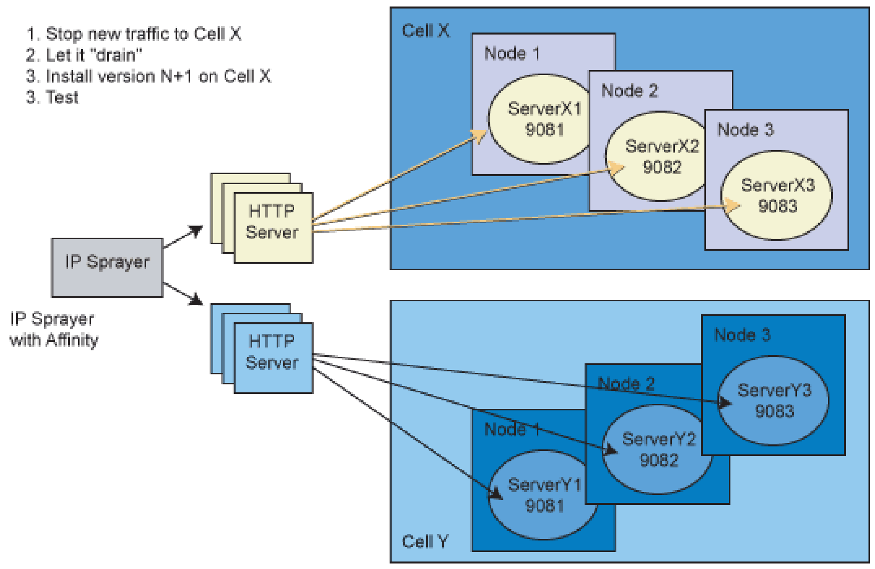
- Cell X, ClusterX -- all servers running application version N
- Cell Y, ClusterY -- all servers running application version N
- Sufficient capacity in either cell to handle full load during the transition period.

B. Begin transition

Figure 3 illustrates the first part of the transition to the new version of the application:

1. Reconfigure the IP sprayer to route all requests not yet associated with a session (that is, new sessions) only to cell Y. Requests associated with existing sessions on cell X continue to flow from the IP sprayer to the HTTP servers associated with cell X.
2. Allow time for the user sessions running on cell X application servers to finish. This is often referred to as "draining" the application servers. After some period of time, all active sessions on cell X servers should have concluded. (The actual amount of time is application-specific.) There are several ways to determine the number of active sessions remaining on the cell X servers:
    - View Performance Monitoring Interface data with Tivoli® Performance Viewer.
    - Monitor the HTTP server access logs.
    - Monitor traffic patterns at the IP sprayer.
3. Install application version N+1 on the cell X application servers.
4. Test cell X version N+1 with a test virtual IP mapped to a test virtual host. This is a final quality assurance test with light load on the application in the production setting. Real load/stress testing was presumably done in a performance test environment that at least mimics the production environment, if not duplicates it. When testing is complete, change the virtual host definition for the Web application to the production virtual host. (This can be done in the WebSphere Application Server admin console or by using a wsadmin script, and requires a restart of the application servers for the change to take effect.)

Figure 3. Transition cell X servers to version N+1 and test

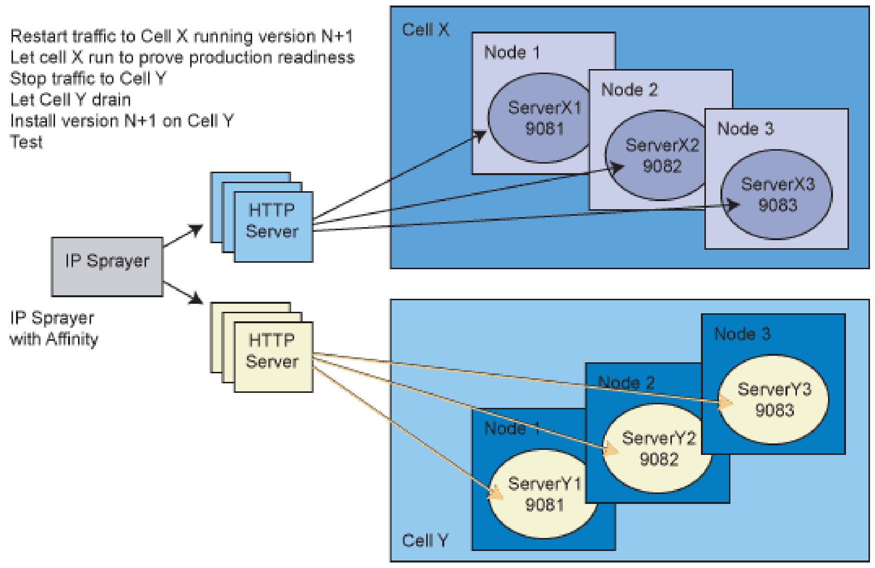


C. Transition to version N+1

Figure 4 illustrates these steps:

1. Reconfigure the IP sprayer to direct requests associated with new sessions to the application servers running version N+1 in cell X. Affinity at the IP sprayer routes requests from version N+1 users back to the servers in cell X, thus continuing to use the same version of the application.
2. Monitor the cell X application servers carefully to ensure proper operation under production load. If failures occur, stop the transition process. Use the IP sprayer to stop additional flow of requests to cell X. Reinstall version N of the application on cell X and open the request flow back up to a fully restored configuration.
3. Once you have confidence that version N+1 is able to operate properly in production, reconfigure the IP sprayer to stop directing new requests to the servers in cell Y. Requests associated with existing sessions on cell Y continue to flow from the IP sprayer to the HTTP servers associated with cell Y so as to not abruptly end those sessions.
4. Let the cell Y servers drain.
5. Install version N+1 of the application on the cell Y servers.
6. Test cell Y with a virtual IP and a test virtual host. (This may be unnecessary depending on your confidence in the installation procedures.) When testing is completed, change the virtual host definition for the Web application to the production virtual host.
7. Reconfigure the IP sprayer to send requests to the HTTP servers associated with cell Y.

Figure 4. Bring cell X, version N+1 on-line; Transition cell Y to version N+1



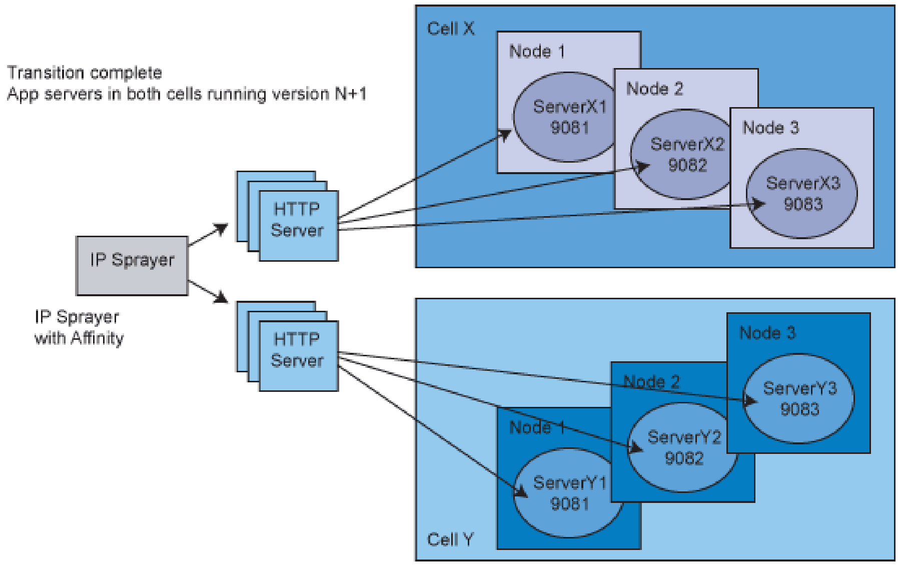
D. Transition complete

The servers in all cells are running version N+1 of the application and all cells are active. Figure 5 illustrates the completed transition with all servers running version N+1 of the application.

With a multi-cell approach:

- There is no need to generate a new `plugin-cfg.xml` or modify it in any manner. The routing tables are dealt with in the IP sprayer, which tends to be easier to administer than modifications to the `plugin-cfg.xml` file
- The procedure is relatively simple and administratively well defined.
- More hardware tends to be needed since some portion of the hardware is taken out of service when that portion is being updated.
- The fallback procedure involves turning off traffic flow to the application servers in the cell running the new version of the application, which is simple enough. However, this relies on the assumption that there is sufficient hardware in the cell running version N of the application to support the site. Getting all servers back to running version N involves an application reinstall on the cell that has been transitioned to version N+1.

Figure 5. Transition complete. Both are cells running version N+1



Single cell approach

Many sites run with only one cell. In a single cell configuration, you can still use multiple clusters to achieve an application rollout while maintaining continuous availability. The essence of the approach is to run with a set of active clusters and a set of dormant (stopped) clusters. In this example, we use only two clusters for the sake of simplicity. The clusters are named ClusterX and ClusterY, with member servers named X1, X2, X3, and Y1, Y2, Y3, respectively. During the rollout period for version N+1 of the application, the dormant clusters are activated, tested and brought online, while the clusters of servers running version N of the application are taken offline, and when all sessions on those servers become inactive, they are shut down.

The servers in the two collections of clusters (active and dormant) are defined on the same hardware. The hardware in use needs to be sufficient to run the load carrying, active servers, as well as the servers that are in the process of being transitioned to an active state. Basically, this means having enough memory for all the JVMs to be running concurrently without paging. The CPU utilization on the servers that are being brought online will ramp up as new sessions get routed to the new version of the application. The CPU utilization on the servers running the old version of the application should ramp down as existing sessions end.

The HTTP plug-in configuration file

To understand the details of the transition procedure, you should have some background on the plug-in configuration file.

One thing of interest in a plug-in configuration file is the ServerCluster element. Below is an example showing the cluster element for ClusterX (modified to show only the relevant attributes and elements, and to fit nicely in the space of this article). The plug-in routes traffic to a server cluster based on the definition of the cluster members, as shown in the listing. If the scope of the `plugin-cfg.xml` file is the entire cell (the usual case), there is a separate ServerCluster element for each cluster defined in the cell. The HTTP plug-in has no other information about the target servers or the mapping of the target applications to those servers other than what is defined in the plug-in configuration file. So even if a collection of servers is defined in a WebSphere cell as being in two separate clusters, it is easy to define them in one cluster from the point of view of the plug-in. It is just a matter of modifying the ServerCluster element to include servers from the other cluster. This is exactly what will be done to seamlessly transition from a cluster of servers running version N of an application to a cluster of servers running version N+1 of an application.

Listing 1. Cluster X cluster element

```
<ServerCluster LoadBalance="Round Robin" Name="ClusterX">
   <Server CloneID="v7oe1ii4" LoadBalanceWeight="2" Name="ServerX1">
     <Transport Hostname="test1.ibm.com" Port="9081" Protocol="http"/>
   </Server>
    <Server CloneID="v7oe1j1e" LoadBalanceWeight="2" Name="ServerX2">
      <Transport Hostname="test2.ibm.com" Port="9082" Protocol="http"/>
    </Server>
    <Server CloneID="v7oe1k2f" LoadBalanceWeight="2" Name="ServerX3">
      <Transport Hostname="test3.ibm.com" Port="9083" Protocol="http"/>
    </Server>
    <PrimaryServers>
      <Server Name="ServerX1"/>
      <Server Name="ServerX2"/>
      <Server Name="ServerX3"/>
    </PrimaryServers>
</ServerCluster>
```

The Server element has a CloneID attribute, mentioned earlier. This value is part of the session cookie created by the Web container in the application servers. For each request that has a session cookie, the plug-in parses out the clone ID from the cookie value and uses it to route the request back to the server where that session is active.

Another important detail of the `plugin-cfg.xml` file is the LoadBalanceWeight attribute of the Server element. As mentioned earlier, this attribute is used to statically assign a weighting factor associated with the round-robin distribution of (new) requests among the servers in a cluster. For the purpose of this article, the important thing to remember about the LoadBalanceWeight is that when it is set to zero, this is a signal to the plug-in to stop sending new requests to that application server -- which is being taken offline. This enables an administrator to gracefully shut down a server. Requests associated with existing sessions on that server will continue to flow to it, but as those sessions get terminated (because the user explicitly logs out or because the session idle timeout is triggered), the server, in time, will no longer have any active sessions. (Monitor the number of active sessions in a server with a lightweight PMI setting and Tivoli Performance Viewer.)

There are other elements in the `plugin-cfg.xml` file that are used to map the application URIs to the clusters that serve them. These elements are depicted below.

Listing 2. Plugin-cfg mapping elements

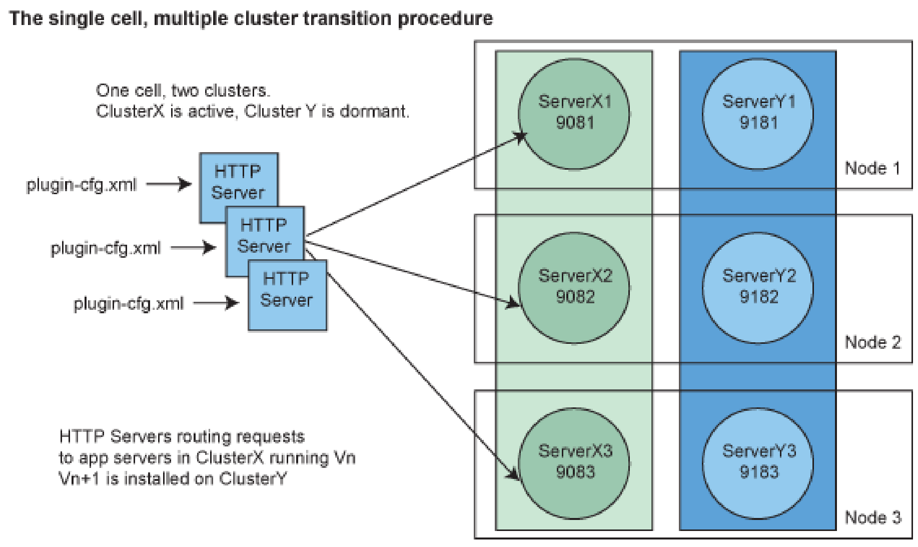
```
<UriGroup Name="prod_vhost_ClusterX_URIs">
        <Uri AffinityCookie="JSESSIONID"
            AffinityURLIdentifier="jsessionid" Name="/user/*"/>
</UriGroup>
<Route ServerCluster="ClusterX"
        UriGroup="prod_vhost_ClusterX_URIs" VirtualHostGroup="prod_vhost"/>

<UriGroup Name="test_vhost_ClusterY_URIs">
        <Uri AffinityCookie="JSESSIONID"
            AffinityURLIdentifier="jsessionid" Name="/user/*"/>
</UriGroup>
<Route ServerCluster="ClusterY" UriGroup="test_vhost_ClusterY_URIs"
          VirtualHostGroup="test_vhost"/>
```

- **UriGroup** -- This element enumerates the URIs for a given application. Often the URI is defined, as shown in Listing 2, by the context root and the asterisk, which matches any remaining portion of the URI beginning with that context root. This particular example is showing there are two applications running that have the same context root of "user." This potential confusion gets resolved by the Route element.
- **Route** -- This element maps the UriGroup to a VirtualHostGroup and a ServerCluster. In this example, with two URI groups that have the same context root, each is using a different VirtualHostGroup, prod_vhost and test_vhost. This means the full URLs will be different in the hostname, port portion of the URL. The URLs with prod_vhost will route to servers in ClusterX, and the URLs with test_vhost will route to servers in ClusterY. (This example is taken from the `plugin-cfg.xml` file for a transition phase where the servers in ClusterX are serving the production application, version N, and the servers in ClusterY are running with version N+1 of the application and being tested through a test virtual IP address mapped to test_vhost.)

The single cell, multiple cluster transition procedure

Figure 6. Single cell, two cluster configuration



A. Starting state

The following is a description of the starting state as depicted in Figure 6:

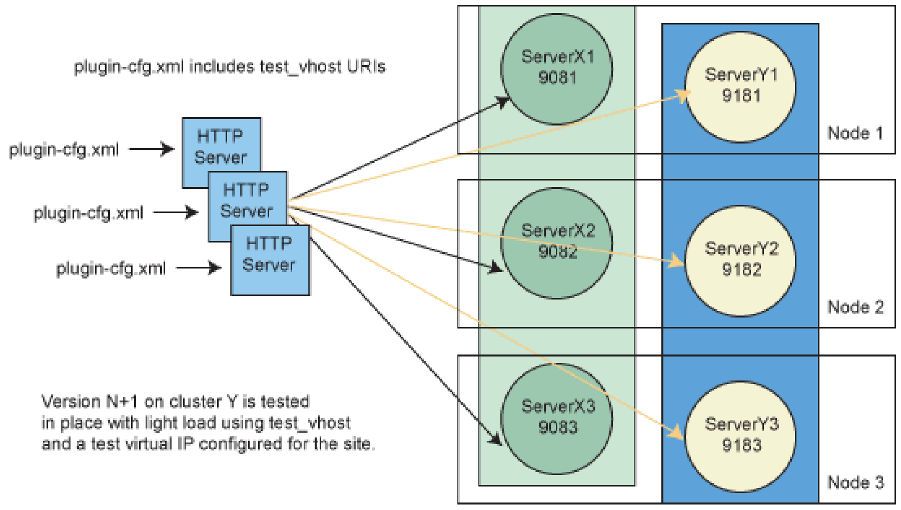
1. Servers in ClusterX are active and running version N of the application.
2. Servers in ClusterY are shut down.
3. The routing table in `plugin-cfg.xml` lists only members of ClusterX as candidates for handling requests.
4. There are two virtual hosts defined:
    - One called prod_vhost for the production version of the application running on the active cluster.
    - One called test_vhost for testing the new version of the application that will be running on the dormant cluster in a final test phase. The test_vhost will need a test virtual IP address assigned to it that can be used by the testers.

B. Begin transition

The following steps describe the transition state, depicted in Figure 7:

1. Install version N+1 of the application on ClusterY:
    a. To make things easy to administer, specify an application name that includes a version number.
    b. Set the virtual host of the Web application to test_vhost. (This can be done at installation time using the admin console or a wsadmin script.)
    c. Dump a new `plugin-cfg.xml` file. This will have both clusters and both versions of the application defined in it. The root context of each Web application will be the same. The routing tables will be separated because of the different virtual hosts the two versions of the application are using.
    d. Copy the `plugin-cfg.xml` file to the HTTP servers. The HTTP servers will automatically reload a new version of the plugin-cfg file at the refresh interval defined in the file itself.
2. Bring up the dormant cluster or some subset of its servers for test purposes. Run a light test on the new version of the application in the production configuration. This is a final test to ensure proper configuration and request routing. These servers are sharing hardware with the production servers actively handling load, and so it is not a good idea to run a heavy test load

Figure 7. Begin transition to version N+1



C. Plug-in configuration file details

At this stage in the process, the `plugin-cfg.xml` file in use by the HTTP servers is as it was generated by the WebSphere GenPluginCfg script or the admin console. It will have two server clusters defined in it, two virtual hosts, and two URI groups, each with the same URI pattern, but each using a different route, since one is using prod_vhost and one is using test_vhost.

D. Transition to version N+1

There are variations on how many nodes to transition. If you want to test just one server running the new version of the application under production load, you can start the transition with just one node. If you feel confident about the new version of the application, or if you are working with multiple clusters, you can transfer all the nodes in a cluster at the same time. One feature of transitioning all the nodes in a cluster is the availability of failover servers running the new version of the application. Depending on the number of servers involved, you may want to transition them in groups of, say, five or so. After the first group is transitioned to a live state and run for some period of time that demonstrates the new version of the application can stand up under production loads, you might transition the remaining servers more quickly:

1. Modify the virtual host mapping of the version N+1 application so that it is using the production virtual host: prod_vhost. ClusterY servers need to be restarted to pick up the change in the virtual host.
2. Move the version N+1 servers (ClusterY) from the dormant cluster into the active cluster (ClusterX):
    a. The nodes that hold two active servers (one running version N of the application, the other running version N+1) are referred to in this description as transition nodes. All three members of the cluster depicted are transitioned at the same time.
    b. The `plugin-cfg.xml` file needs to be modified as described below in Listing 3. In our example, the servers in ClusterY get moved into ClusterX.
    c. The LoadBalanceWeight of the ClusterX servers is set to 0 so the plug-in will not send them any more requests that are not already associated with a session. These servers will only receive requests associated with existing sessions tied to those servers by session affinity.
3. Copy the `plugin-cfg.xml` file to the HTTP servers. Once the HTTP server plug-ins re-read this updated copy of `plugin-cfg.xml`, the routing tables will include ClusterY servers (running version N+1 of the application) in the production collection of servers. (See Listing 3.) Requests not yet associated with an active session ("new" requests) will get routed to the ClusterY servers in the normal round robin fashion.

E. Plug-in configuration file details

At this point, the ServerCluster element for ClusterX in the `plugin-cfg.xml` file looks like Listing 3. The LoadBalanceWeight for all the actual members of ClusterX is set to 0. All the actual members of ClusterY have been moved to the ClusterX element and all have a positive LoadBalanceWeight. Not shown in the listing is the fact that the UriGroup and Route elements for the application actually installed on ClusterY (version N+1 of the application) have been removed from the `plugin-cfg.xml` file. In addition, the ServerCluster element for ClusterY has been removed.

Listing 3

```
<ServerCluster LoadBalance="Round Robin" Name="ClusterX">
  <Server CloneID="v7oe1ii4" LoadBalanceWeight="0" Name="ServerX1">
    <Transport Hostname="test1.ibm.com" Port="9081" Protocol="http"/>
  </Server>
  <Server CloneID="v7oe1j1e" LoadBalanceWeight="0" Name="ServerX2">
    <Transport Hostname="test2.ibm.com" Port="9082" Protocol="http"/>
  </Server>
  <Server CloneID="v7oe1k2f" LoadBalanceWeight="0" Name="ServerX3">
    <Transport Hostname="test3.ibm.com" Port="9083" Protocol="http"/>
  </Server>
  <Server CloneID="v7oe3bhc" LoadBalanceWeight="2" Name="ServerY1">
    <Transport Hostname="test1.ibm.com" Port="9181" Protocol="http"/>
  </Server>
  <Server CloneID="v7oe3c36" LoadBalanceWeight="2" Name="ServerY2">
    <Transport Hostname="test2.ibm.com" Port="9182" Protocol="http"/>
  </Server>
  <Server CloneID="v7oe4cu8" LoadBalanceWeight="2" Name="ServerY3">
    <Transport Hostname="test3.ibm.com" Port="9183" Protocol="http"/>
  </Server>
  <PrimaryServers>
      <Server Name="ServerX1"/>
      <Server Name="ServerX2"/>
      <Server Name="ServerX3"/>
      <Server Name="ServerY1"/>
      <Server Name="ServerY2"/>
      <Server Name="ServerY3"/>
  </PrimaryServers>
</ServerCluster>
```

F. Transition complete

In the final state, the members of ClusterX have been shut down. (They are left running long enough to become inactive and long enough to be convinced that a rollback is not going to be needed.)

With a single cell, multiple cluster approach:

- You get efficient use of the available hardware because you do not have to be able to support the full load of the site on some portion of the hardware.
- There is somewhat complicated administrative "trickery" involved with the `plugin-cfg.xml` file.
- There is a good rollback procedure since the all application servers that you would roll back to still have version N of the application installed on them. The rollback procedure involves using the starting state version of the `plugin-cfg.xml` file.

Variations on a theme

Now that you've seen the multi-cell and single cell approaches to an application rollout, you can probably imagine the potential for variations that can lead to a procedure that fits your particular situation better than what has been specifically described here.

One variation on the multiple cell approach is to use the same physical hardware for both cells, with one cell being active and one being dormant (stopped). This rollout procedure will just be summarized here, but it will look a lot like the single cell, multi-cluster approach described earlier, with respect to `plugin-cfg.xml` file manipulation. During the rollout period, the dormant cell is brought up and the new version of the application is installed on it. Testing can be done by merging plug-in configuration files from the two cells, using two clusters (one from each cell), but using a test virtual host and virtual IP address for version N+1 of the application. With this approach, the hardware needs to be sufficient to support two node agents and two application servers per machine running concurrently during the rollout period. The transition period involves constructing a merged `plugin-cfg.xml` file with servers from both cells, defined as part of a single cluster element, and the LoadBalanceWeight of the version N servers set to 0 so they can drain. Once the version N servers have gone idle, the version N cell can be shut down. The `plugin-cfg.xml` file generated from the version N+1 cell can then be used to configure the HTTP server plug in, and then the transition is complete.

Java client application transition

Next, we will look at a procedure for managing the transition of Java client applications from a current version to a new version. Java client transition is not usually a major topic of interest, since Java clients are not commonly developed; when they are, there tends to be sufficient administrative control over the client population making it feasible to distribute a new client the same time the new version of the application goes live. However, should a more large scale Java client scenario be involved, the process described below enables a smooth transition with relatively simple administrative tasks.

In this process:

- It is assumed that the EJB methods involved with the client API are either unchanged or backward compatible with existing client applications.
- Our example has only a single EJB component used for the client API, named SessionFacade. The name of the simple application is userHitCount.
- This approach assumes a single cell with multiple clusters defined, one active and one dormant, as shown in Figure 8.

The phrase "Java client" refers to a standalone Java client, not a client running in an application server. The case where the EJB client is running on an application server can be treated pretty much the same as what is described here, with a simplification: there is no need to be concerned about which initial context provider URL to use. Anything running in an application server can use the default initial context provider.

For new application version rollout, there are two issues that arise with respect to stand-alone Java clients:

- **Where to get the initial context**
  The initial context provider URL is usually defined as a configuration parameter or command line input for the client. Unfortunately, you do not have access to change the provider URL if a new server cluster is going to be providing the new version of the application

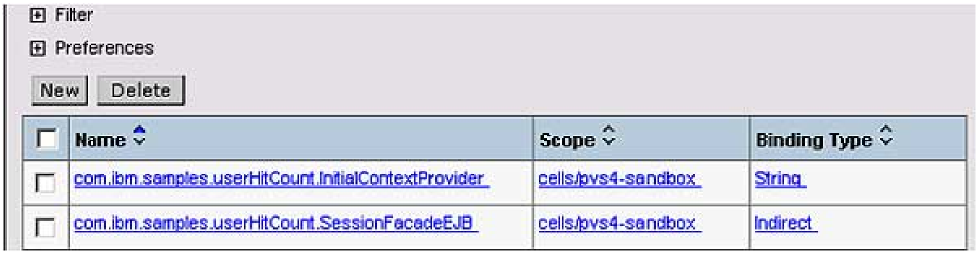
- **EJB bindings**
  These are defined in the deployment descriptor of the client. If you do not have access to the client executable, you cannot change these.

With WebSphere Application Server V5, there is a feature referred to as configured bindings in JNDI that can be used to handle both of the above issues. A cell persistent string binding can be created with the initial context provider URL. A cell persistent indirect binding can be used to get to the EJB servers running the current version of the application.

Configured JNDI entries can be defined and managed from the admin console under **Environment => Naming => Namespace Bindings**. They can also be configured using a wsadmin script.

The standalone Java client is programmed to use a "bootstrap" initial context provider that points to the node agents of the cell (which are intended to be always running so they can be relied upon to always be available). A lookup is done for a runtime initial context provider URL that points to the servers in the cluster running the latest version of the application. (The details will be clear from the Listing 4.)

Figure 8. Admin console view of configured JNDI bindings for Java client use



	Name	Scope	Binding Type
☐	com.ibm.samples.userHitCount.InitialContextProvider	cells/pvs4-sandbox	String
☐	com.ibm.samples.userHitCount.SessionFacadeEJB	cells/pvs4-sandbox	Indirect

The values of the configured bindings are switched when the transition is made:

Listing 4

```
InitialContextProvider (when ClusterX is active, note the odd port #s)
corbaloc::test1.ibm.com:9811,:test2.ibm.com:9813,:test3.ibm.com:9815

InitialContextProvider (when ClusterY is active, note the even port #s)
corbaloc::test1.ibm.com:9812,:test2.ibm.com:9814,:test3.ibm.com:9816

EJB indirection JNDI binding when ClusterX is active:
  com/ibm/samples/userHitCount/SessionFacadeEJB
 Provider URL (odd port #s):
  corbaloc::test1.ibm.com:9811,:test2.ibm.com:9813,:test3.ibm.com:9815
 Binding: cell/clusters/ClusterX/ejb/com/ibm/samples/uhc/ejbs/SessionFacadeHome

EJB indirection JNDI binding when ClusterY is active:
  com/ibm/samples/userHitCount/SessionFacadeEJB
 Provider URL (even port #s):
  corbaloc::test1.ibm.com:9812,:test2.ibm.com:9814,:test3.ibm.com:9816
 Binding: cell/clusters/ClusterY/ejb/com/ibm/samples/uhc/ejbs/SessionFacadeHome
```

The client code will have a local resource environment reference:

```
java:comp/env/myInitialConext
```

and a local EJB reference:

```
java:comp/env/ejb/mySessionFacade
```

The standalone Java client will have two bindings in its deployment descriptor:

```
java:comp/env/myInitialContext
```

Binding:

```
cell/persistent/com/ibm/samples/userHitCount/InitialContextProvider
```

and:

```
java:comp/env/mySessionFacade
```

Binding:

```
cell/persistent/com/ibm/samples/userHitCount/SessionFacadeEJB
```

The bindings used in the client deployment descriptor must match the JNDI name specified for the configured bindings. In this example, the scope of the configured bindings is the cell persistent (cell/persistent) portion of the name space. The remaining portion of the JNDI name is clearly specific to the example and is made up of a package and application name.

The bootstrap initial context provider URL has a list of node agents, since node agents are always running regardless of which server cluster is currently active. This provider URL is provided on the command line of the Java client, or as a property from a configuration file. The client has a method to randomize the order of the providers in the URL to avoid having all clients use the same node agent or application server as their name service provider. (In large scale Java client situations, that can lead to an overloaded process.)

The code in the Java client to do the initial lookup looks as follows:

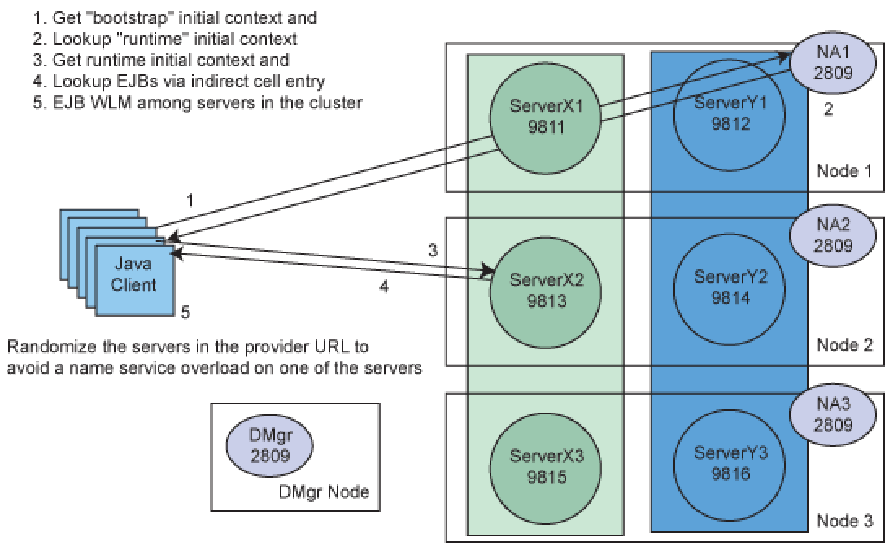
Listing 5

```
try
{
 // The url is a parameter from the command line
 String randomizedURL = randomizeURL(url);
 env.setProperty(Context.PROVIDER_URL, randomizedURL);
 env.setProperty(Context.INITIAL_CONTEXT_FACTORY,DEFAULT_FACTORY);
 // Get the "bootstrap" context
 bootstrap_ctx = new InitialContext(env);

 // lookup the "runtime" context
 myProviderURL = (String)
  bootstrap_ctx.lookup("java:comp/env/myProviderURL");
 randomizedURL = randomizeURL(myProviderURL);
 env.setProperty(Context.PROVIDER_URL, randomizedURL);

 // turn off caching for JNDI lookups, to avoid stale home issues.
 env.setProperty(PROPS.JNDI_CACHE_OBJECT,
  PROPS.JNDI_CACHE_OBJECT_NONE);
 runtime_ctx = new InitialContext(env);
}
```

Figure 9. Starting state for Java application update

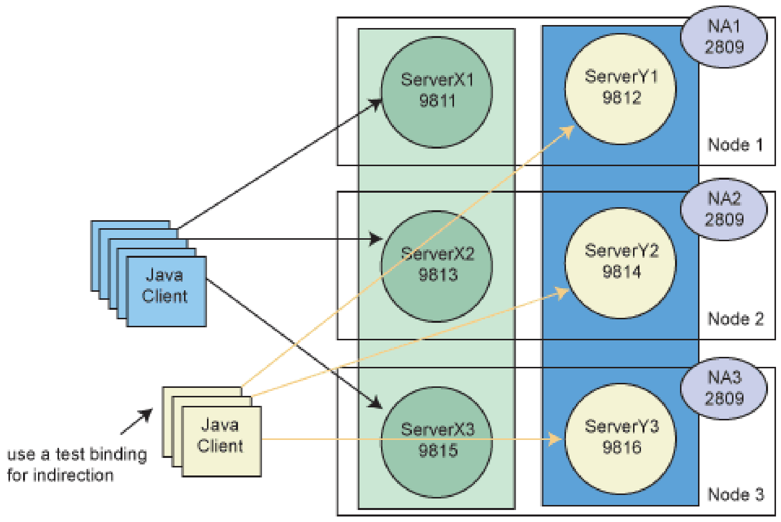

1. Get "bootstrap" initial context and
2. Lookup "runtime" initial context
3. Get runtime initial context and
4. Lookup EJBs via indirect cell entry
5. EJB WLM among servers in the cluster
Java Client
1
2
3
4
5
Randomize the servers in the provider URL to avoid a name service overload on one of the servers
DMgr 2809
DMgr Node
ServerX1 9811
ServerY1 9812
NA1 2809
Node 1
ServerX2 9813
ServerY2 9814
NA2 2809
Node 2
ServerX3 9815
ServerY3 9816
NA3 2809
Node 3

A. Starting state

The starting state is depicted in Figure 9:

1. Servers in ClusterX are running version N of the application.
2. Servers in ClusterY are stopped.
3. Configured bindings in JNDI point to servers in ClusterX.
4. Java clients are work load managed among ServerX members.

Figure 10. Testing version N+1 on ClusterY



B. Begin transition

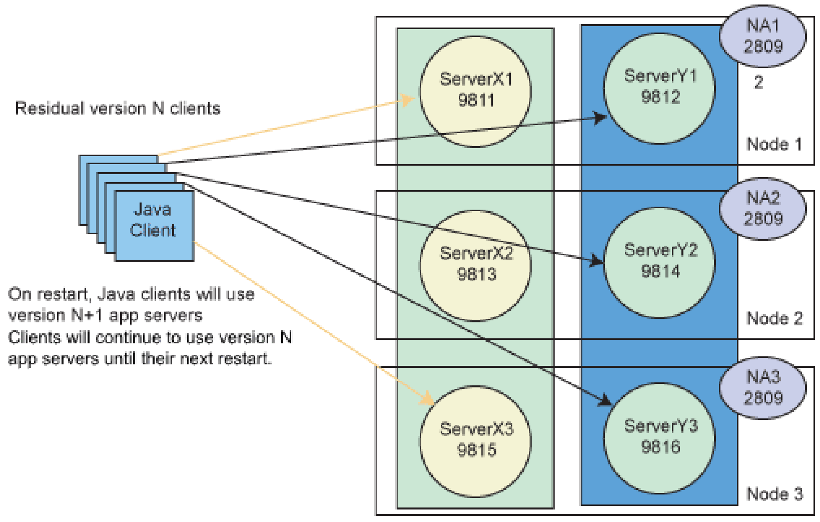
At this point, version N+1 of the application is installed on ClusterY and the servers are started up to provide a final test environment on the production hardware. Some test Java clients can be configured to use test bindings that point to the ClusterY servers. The test load should be kept minimal since the ClusterY servers are sharing the same hardware as the production servers in ClusterX.

C. Transition to version N+1

Transition occurs all at once. There is no direct access to the EJB routing table as there is with the Web client routing table in `plugin-cfg.xml`, so the only transition choice is all servers at once:

1. Modify the two configured bindings: InitialContextProvider string binding and the EJB indirection binding to point to cluster N+1 (in this case ClusterY).
2. The cluster N+1 (ClusterY) servers will need to be stopped and started to pick up this change in the cell persistent namespace.
3. On the next restart of any of the Java clients, they will pick up the new bindings and start working with version N+1 of the application.
4. The version N servers are left running until all EJB activity has subsided to the point where they can be shut down. The administrator can use netstat to monitor the number of ORB port connections to the version N servers to determine activity level. Alternatively, Tivoli Performance Viewer can be used to monitor EJB activity, but the PMI for watching EJB method call activity may introduce more overhead than desired.

Figure 11. Transition to versoin N+1 servers


Residual version N clients
Java Client
On restart, Java clients will use version N+1 app servers
Clients will continue to use version N app servers until their next restart.
ServerX1 9811
ServerY1 9812
NA1 2809
2
Node 1
ServerX2 9813
ServerY2 9814
NA2 2809
Node 2
ServerX3 9815
ServerY3 9816
NA3 2809
Node 3

D. Transition complete

Once it has been determined that there is no (or very little) activity on the version N servers, they are shut down. In the case where servers are being used for Web clients and Java clients, you would need to check for activity in the Web application as well

Conclusion

This article presented detailed approaches to performing an application update in a production setting while maintaining continuous availability of the application. Perhaps the simplest approach uses server clusters in multiple cells all running on their own hardware. However, that approach can make inefficient use of the available hardware. An administratively more complex approach that makes efficient use of the hardware is to use an active/dormant cluster (or cell) pair defined to use the same hardware. During the transition from version N to version N+1 of the application, both sets of servers are running, while the version N set is ramping down and the version N+1 set is ramping up. Assuming all goes well, when the version N servers are drained of activity, they are shut down. These procedures offer a fast and easy rollback option should the need arise to revert to the previous version of the application. Web client and Java client scenarios were both discussed.

Acknowledgements

Several consultants provided input for this article:

- Randy Powelson, Premier ITC
- Renuka Chekkala, IBM Software Services for WebSphere
- Ben Neu, IBM Software Services for WebSphere
- Keys Botzum, IBM Software Services for WebSphere
- Jack Gibson, Brulant.

Related topic

- To learn more about WebSphere Application Server, visit the developerWorks WebSphere Application Server zone. You'll find technical documentation, how-to articles, education, downloads, product information, technical support resources, and more.