**Module 13**

**Tomcat Clustering**

In this module, we detail the process of clustering Tomcat: setting up multiple machines to host your web applications. There are several significant problems related to running your web application on a single server. When your web site is successful and begins to get a high volume of requests, eventually one server computer just won’t be able to keep up with the processing load. Another common problem of using a single server computer for your web site is that it creates a single point of failure. If that server fails, your site is immediately out of commission. Regardless of whether it’s for better scalability or for fault tolerance, you will want your web applications to run on more than one server computer. This chapter will show you how to set up a clustered Tomcat system that does exactly that.

Clustering is an advanced topic and is not useful to everyone. Also, as of the early stable versions of Tomcat 6.0, the code that makes clustering possible is still somewhat immature and should be considered experimental code, unless you exhaustively test your installation and find it to be stable. You should perform your own testing to ensure that clustering works in your environment.

Giving all of the details of clustering techniques, or even exhaustively covering how a particular clustering product works, is beyond the scope of this book. There are numerous ways to cluster any network service, but we can show you only a couple of popular examples. However, this chapter will give you some ideas about hardware and software that you can use, how clustering generally works, and how you can configure Tomcat for some clustering use cases. Be sure to see the [“Additional Resources”](#_bookmark806) section, at the end of this module, for URLs to many open source project web sites, where you can find more detailed documentation on how to install and configure the software packages mentioned in this module.

**Clustering Terms**

Before we dig into the details about how to set up a Tomcat cluster, we want to be clear on the definitions of some terms we’ll be using in this chapter:

**Fault tolerance**

The degree to which the server software adapts to failures of various kinds (including both hardware and software failures) so that the system may still serve client requests transparently, usually without the client being aware of these failures.

**Failover**

When one server (software or hardware) suffers a fault and cannot continue to serve requests, clients are dynamically switched over to another server that can take over where the failed server left off.

**High availability**

A service that is always up, always available, always serving requests, and can serve an unusually large volume of requests concurrently is said to be highly available. A highly available service must be fault tolerant, or else it will eventually fail to be available due to hardware or software failures.

**Distributed**

The term “distributed” simply means that some computing process may occur across multiple server computers working together to achieve a goal or to formulate an answer or multiple answers, ideally in parallel. For example, many web server instances each running on a separate server computer behind a TCP load balancer constitutes a distributed web server.

**Replicated**

Replication means that any state information is copied verbatim to two or more server software instances in the cluster to facilitate fault tolerance and distributed operation. Usually, stateful services that are distributed must replicate client session state across the server software instances in the cluster.

**Load balancing**

When a request is made to a distributed service and the server instance that received the request is too busy to serve it in a reasonable amount of time, another server instance may not be as busy. A load-balanced service is able to forward the request to a less busy server instance within the cluster to be served. Load balancing can distribute the request-processing load to take advantage of all available computing resources.

**Cluster**

A cluster is made up of two or more server software instances running on one or more server computers that work together to transparently serve client requests so that the clients perceive the group as a single highly available service. The goal of the group is to provide a highly available service to network clients, while utilizing all available computing resources as efficiently as reasonably possible.

In general, clustering exists to facilitate high availability and/or fault tolerance. Load balancing and state replication are just two important elements of clustering. Clustering may be done in a simple way so that the requests are distributed among server software instances within the cluster that aren’t aware of each other, or it may be implemented in a tightly integrated way such that all server software instances within the cluster are aware of each other and replicate state among each other.

**The Communication Sequence of an HTTP Request**

To configure and run a Tomcat cluster, you need to set up more than just Tomcat. For example, you need to provide a facility so that requests coming into Tomcat are spread across multiple instances. This involves software that runs in addition to your Tomcat installations.

To identify the points in the system where clustering features may be implemented to distribute the requests, let’s take a look at the steps of the average HTTP client request. [Figure 13-1](#_bookmark1561) shows a typical nonclustered server running Apache httpd, mod\_ proxy, and Tomcat. The figure shows the steps of one HTTP client’s request through the system.



Corporate/private network

**2**

Local DNS server

Public Internet

Authoritative DNS server

**3**

**1**

**4**

**5**

Web browser

Public Internet

Apache

*httpd*

*mod\_proxy* (or other connector)

**6**

**7**

Tomcat DB

User

*Figure* *13-1. How one HTTP request uses a typical nonclustered server*

We show using *mod\_proxy* for the connector module from Apache httpd to Tomcat because depending on how your web application is written, you may not need to use Apache httpd or *mod\_proxy* to set up anduse a Tomcat cluster. We show these components so that you can see how using them affects the HTTP request communication sequence and which types of clustering features you may want to use. If you use Apache httpd, httpd is your web server. If you use Tomcat standalone, Tomcat is your web server.

Any user’s HTTP request to the server follows these steps:

1. Local DNS request. The user’s web browser attempts to resolve the web site’s IP address from its name via a DNS lookup network request to the user’s local DNS server (usually her ISP’s DNS server or her own company’s DNS server). Most web browsers ask for this IP address only once per run of the browser. Subsequent HTTP requests from the same browser are likely to skip this step as well as the next step.
2. Authoritative DNS request. Usually, the user’s local DNS server will not already have the web site’s IP address in its cache (from a prior request), so it must in turn ask the web site’s authoritative DNS server for the IP address of the web site that the user wishes to view. The authoritative DNS server will reply to the local DNS server with the IP address that it should use for the web server. The local DNS server will attempt to cache this answer so that it won’t need to make the same request to the authoritative DNS server again anytime soon. Subsequent requests from other browsers in the same network as the first browser are likely to skip this step because the local DNS server will already have the answer in its cache.
3. Local DNS response. The local DNS server replies, giving the browser the IP address of the web server.
4. HTTP request. The browser makes an HTTP request to the IP address given by DNS. This request may utilize HTTP keep-alive connections for network effi- ciency, and therefore this single TCP socket connection may be the only socket connection made from the browser to the web server for the entire duration of the browser’s HTTP session. If the browser does not implement or use HTTP keep-alive, each request for a document, image, or other content file will create a separate TCP socket connection into the web server.
5. Tomcat sends one or more requests to backend server(s). Tomcat may depend on other servers to create the dynamic content response that it forwards back to the browser. It may connect to a database by way of JDBC, or it may use JNDI to look up other objects, such as Enterprise JavaBeans, and call one or more methods on the objects before being able to assemble the dynamic content that makes up the response.
6. Upon completion of the necessary steps above, the direction of flow reverses and replies to each step are made in the reverse order that the request steps were made, working back through the already open network connections.

For your cluster to be fault tolerant, so that it is still 100 percent useable when any single hardware or software instance fails, it must have no single point of failure. You must have two or more of each component that is necessary to process any request. For instance, if you are using Apache httpd in front of Tomcat, you can’t set up just one Apache httpd and two Tomcat instances behind it because if httpd fails, no requests will ever make it to any of the Tomcat instances. In that case, Apache httpd is a single point of failure.

To support a cluster of Apache httpd andTomcat instances, you can implement clustering features in multiple spots along this request sequence. [Figure 13-2](#_bookmark1567) shows the same request sequence, only this time the web site is served on a cluster of Apache httpd and Tomcat instances.



Corporate/private network

Local DNS server

Public Internet

Authoritative DNS server

Apache

*httpd* 1

*mod\_proxy*

Tomcat 1

Web browser

Public Internet

DB

User

Apache

*httpd* 2

*mod\_proxy*

Tomcat 2

*Figure 13-2. A request through a cluster of Apache httpd and Tomcat instances*

Here are some of the clustering technologies that you could set up and run:

***DNS request distribution***

Instead of configuring your DNS server to give out one IP address to one Apache httpd server instance, you can configure it to give out three IP addresses that each go to a separate Apache httpd or Tomcat instance.

***TCP Network Address Translation (NAT) request distribution***

Regardless of how many IP addresses DNS gives to the client’s browser, the web server’s first contact IP address(es) can be answered by a TCP NAT request distributor that acts as a gateway to two or more web servers behind it. You can use the NAT request distributor for both load balancing and failover.

***mod\_proxy\_balancer load balancing and failover***

If you run two or more Tomcat instances behind one or more Apache httpd instances, you can use mod\_proxy\_balancer for loadbalancing andfailover to distribute requests across your Tomcat cluster. You can also use it to keep requests from being distributed to any failed Tomcat instances.

***JDBC request distribution and failover***

You could use a clustered database and a JDBC driver that load balances connections among the machines in the database cluster or a replicated database with a JDBC driver that knows when to failover to the secondary database server.

### **DNS Request Distribution**

Request distribution can be done at the authoritative DNS server. This is a Wide Area Network (WAN) clustering solution that can distribute requests across server machines at one or more data centers. If you do not have authoritative control for at least one fully qualified hostname in your domain and can use at least two static IP addresses, you cannot take advantage of DNS request distribution. You may, however, be able to take advantage of other request distribution methods.

When the browser’s local DNS asks for an IP address from the web site’s authoritative DNS, and there are two machines in the cluster that run web servers, which IP address should the authoritative DNS reply with? DNS can give multiple answers to a single question—it can give both IP addresses to the browser, but the browser will use only one of the addresses.

Most of the time, system administrators set up general-purpose DNS server software (such as BIND, for example) for their authoritative DNS servers, and any local DNS asking for the IP address to the cluster of web servers will be given all of the IP addresses that are mapped to the web server hostname. It’s up to the browser to choose which of the returned addresses to use. The browser typically uses the first address in the list of addresses given to it by its local DNS.

To balance the load a bit, most DNS server software will give out the list of IP addresses in a different, circular order every time a request is made. This means that no specific IP address stays at the top of the list, and therefore the browsers will use the IP addresses in a circular order. This is commonly known as DNS *round-robin*. DNS round-robin is simple and relatively easy to configure, but it has many drawbacks.

The best you can hope for is random distribution of requests among all of the servers in the cluster because of DNS caching and varying browser implementations. Usually, the distribution is random, but there is no guarantee that it will be evenly random. Although DNS round-robin can break up requests to different server machines in the cluster, that doesn’t mean that there won’t be times when one server machine gets most of the cluster’s load. The more a service needs to scale, the larger this problem becomes.

***It does not take load into account***

General purpose DNS software such as BIND isn’t written to know anything about content server load. So, round-robin will eventually send clients to a server machine that is overloaded, resulting in failed requests.

***It is not fault tolerant***

It won’t know anything about machines that are down or have been temporarily removed from the cluster’s service pool, so round-robin will eventually send clients

to a server machine that is down. If an online store’s web site has 10 machines in the cluster and one machine goes down, 10 percent of the purchases (and the revenue for those purchases) are lost until an administrator intervenes.

***It knows nothing about congested networks, nor downed network links***

If the authoritative DNS is providing IP addresses to server machines residing in two different data centers, and the high-bandwidth link to the first data center goes down, DNS round-robin may in fact send half of a web site’s clients to unreachable IP addresses.

To do load balancing with DNS without the problems of DNS round-robin, the DNS software must be specially written to monitor things such as server load, congested or down network links, down server machines, and so on. Smart DNS request distributors such as Citrix Netscaler ([*http://www.netscaler.com*](http://www.netscaler.com/)), Foundry Networks’ ServerIron ([*http://www.foundrynet.com/solutions/sol-app-switch*](http://www.foundrynet.com/solutions/sol-app-switch)), and Cisco’s DistributedDirector ([*http://www.cisco.com/en/US/products /hw/contnetw/ps813/index.*](http://www.cisco.com/en/US/products%20/hw/contnetw/ps813/index.)[*html*](http://www.cisco.com/en/US/products/hw/contnetw/ps813/index.html)) can be configured to monitor many metrics (including server load) and use them for request distribution criteria. For instance, if one of the data centers loses connectivity to the public Internet, these smart DNS request distributors could monitor the link and be aware of the outage and not distribute any requests to those servers until the link is working again. With such great fault tolerance features, DNS request distribution is an excellent way to initially distribute your request load.

### **TCP NAT Request Distribution**

Once DNS has given the user’s web browser at least one IP address, the web browser opens a TCP connection to that IP address. The web browser will send an HTTP request over this TCP socket connection. In a non clustered setup, this IP address goes to the one and only web server instance (it could be Tomcat’s web server, or Apache httpd, or even some other HTTP server implementation). But, in a clustered environment, you should be running more than one web server instance, and requests should be balanced across them. You may use a DNS request distributor to distribute requests directly to these web server instances, or you can point DNS to a TCP Network Address Translation (NAT) request distributor, which will distribute requests across your web servers.

[Figure 13-3](#_bookmark1588) shows a NAT request distributor in front of three web server instances, each on its own server computer.

NAT request distributors may be used for load balancing, fault tolerance, or both. When a browser makes a TCP connection to the NAT request distributor, it can use one of many possible request distribution algorithms to decide which internal web server instance to handoff the connection to. When you initially set up andconfigure a NAT request distributor, you will choose the algorithm you want to use for distributing requests. The available algorithms vary with the different NAT request

Corporate/private network

Local DNS server

Public Internet

Authoritative DNS server

server computer 1

web server 1

Web browser

Public Internet

NAT request distributor

server computer 2

web server 2

User

server computer 3

web server 3

*Figure 13-3. A TCP NAT request distributor distributing an HTTP request*

distributor implementations. Generally, all distributors will offer at least a round- robin algorithm. Some can monitor the load on the web server machines and distribute requests to the least loaded server, and some allow the administrator to give each web server machine a weighted value representing the capacity of each server and distribute requests based on the relative capacity differences.

Most NAT request distributors also offer fault tolerance by detecting various kinds of web server faults and will stop distributing requests to any server that is down. For example, in [Figure 13-3,](#_bookmark1588) if web server 2’s operating system crashes and does not reboot on its own, the NAT request distributor will stop distributing requests to web server 2, and will evenly balance all of the request load across web servers 1 and 3. The users of the site won’t notice that web server 2 has crashed and may continue using the site while the system administrator reboots web server 2’s machine and brings it back online. Once server 2 is back, the NAT request distributor will automatically notice that it’s back and will resume sending requests to it.

There are many NAT request distributor implementations available, both commercially and as open source software that runs on commodity computer hardware. Here are several free implementations:

***Linux IPTables***

On Linux, the 2.6.x kernels come with a facility called IPTables that is able to per- form various kinds of network packet filtering, translation, and forwarding, including NAT and load balancing. This is helpful because if you’re running Linux, you already have it installed, so you don’t need to build or install anything extra. You just need to configure it. Some of the best documentation for this is the iptables manpage (type *man iptables*).

***The*** ***Linux Virtual Server Project’s VS-NAT***

The Linux Virtual Server Project ([http://www.linuxvirtualserver.org](http://www.linuxvirtualserver.org/)) distributes an open source (free) software package called VS-NAT that runs only on the Linux operating system, but is feature-rich, and comes with good documentation. See <http://www.linuxvirtualserver.org/VS-NAT.html> for details.

***IP Filter***

This is another open source software package and runs on most UN\*X-like operating systems, with the apparent exception of Mac OS X. It is used for many packet-filtering purposes, but it can be used as a round-robin NAT request distributor as well. The IPF home page is [http://cheops.anu.edu.au/~avalon](http://cheops.anu.edu.au/%7Eavalon), and you can find information about how to use it as a request distributor at [http://www.](http://www.obfuscation.org/ipf/ipf-howto.html#TOC_38) [obfuscation.org/ipf/ipf-howto.html#TOC\_38](http://www.obfuscation.org/ipf/ipf-howto.html#TOC_38).

### ***mod\_proxy Load Balancing*** ***and Failover***

If you decide to use Apache httpd as your web server, and you’re using mod\_proxy to send requests to Tomcat (either via HTTP or AJP), you can take advantage of *mod\_ proxy\_balancer’s* load balancing and fault tolerance features. These Apache httpd modules are part of the Apache httpd web server project, and you’ll be happy to know that the *mod\_proxy\_\** modules are almost always built and shipped with Apache httpd. This means that you can configure and use them without downloading, building, and installing anything extra, which makes it quite a bit easier to set up and use a load balancer.

Here are some of the things that each Apache httpd with *mod\_proxy\_balancer* in your cluster can do:

***Distribute requests to one or more Tomcat instances***

You can configure many Tomcat instances in your Apache httpd’s configuration, giving each Tomcat instance an lb\_factor value that functions as a weighted request distribution metric.

***Detect Tomcat instance failure***

*mod\_proxy\_balancer* will detect when a Tomcat instance’s connector service is no longer responding and will stop sending requests to it. Any remaining Tomcat instances will take the additional load for the failed instance until it is brought back online.

***Detect when a Tomcat instance comes back up after failing***

After *mod\_proxy\_balancer* has stopped distributing requests to a Tomcat instance due to the instance’s connector service failure, *mod\_proxy\_balancer* periodically checks to see if the server is available again and will automatically converge it into the pool of active Tomcat instances when it becomes available again.

***Manually mark a*** ***Tomcat instance available or unavailable***

During the runtime of Apache httpd you may use the *balancer-manager* web page (a feature built into Apache httpd, implemented by *mod\_proxy\_balancer* and *mod\_status*) to mark Tomcat instances as being either available or unavailable. This allows adding or removing clustered Tomcat instances without any web site down time as it does not require restarting Apache *httpd* for the change to take effect. Again, the good news here is that these modules are almost always built into Apache httpd, andall you need to do to use these features is configure your Apache *httpd* to turn it on.

The following steps outline how to set up one Apache httpd (on a server computer called apache1) to do HTTP load balancing across two Tomcat instances that reside on two separate server computers called tc1 and tc2.

First, add the following configuration to your Apache httpd’s configuration files (we added it in the form of a new config file named /etc/httpd/conf.d/proxy-balancer.conf, but you may need to place it in a different file for your installation of Apache httpd):

*<IfModule !proxy\_module>*

*LoadModule proxy\_module modules/mod\_proxy.so*

*</IfModule>*

*#<IfModule !proxy\_ajp\_module>*

*# LoadModule proxy\_ajp\_module modules/mod\_proxy\_ajp.so*

*#</IfModule>*

*<IfModule !proxy\_http\_module>*

*LoadModule proxy\_http\_module modules/mod\_proxy\_http.so*

*</IfModule>*

*<IfModule !proxy\_balancer\_module>*

*LoadModule proxy\_balancer\_module modules/mod\_proxy\_balancer.so*

*</IfModule>*

*<IfModule !status\_module>*

*LoadModule status\_module modules/mod\_status.so*

*</IfModule>*

*<IfModule proxy\_balancer\_module> ProxyRequests off*

*<Proxy balancer://tccluster>*

*BalancerMember http://tc1:8080 loadfactor=1 max=150 smax=145 BalancerMember http://tc2:8080 loadfactor=1 max=150 smax=145 Order Deny,Allow*

*Allow from all*

*</Proxy>*

*<Location /balancer-manager> SetHandler balancer-manager Order Deny,Allow*

*Allow from all*

*</Location>*

*<Location /my-webapp>*

*ProxyPass balancer://tccluster/my-webapp stickysession=jsessionid ProxyPassReverse balancer://tccluster/my-webapp*

*Order Deny,Allow Allow from all*

*</Location>*

*<Location /examples>*

*ProxyPass balancer://tccluster/examples stickysession=jsessionid ProxyPassReverse balancer://tccluster/examples*

*Order Deny,Allow Allow from all*

*</Location>*

*</IfModule>*

This configuration will load balance two Tomcat instances running on two separate hosts (named tc1 and tc2). The load will be distributed evenly between both Tomcat instances, but once Tomcat creates a session for the client and sends the client a *JSESSIONID* cookie, *mod\_proxy\_balancer* will distribute that client’s requests to the same Tomcat instance each time. The above configuration proxies the /my-webapp and /examples base URIs through to the cluster of Tomcat instances, so that requests for those webapps are handled by the Tomcat cluster. The configuration also turns on the /balancer-manager page so that the cluster instances may be managed via a web browser.

You can set the *loadfactors* to any integer values you want. The higher the number you use, the more preferred the Tomcat instance is; the lower the *loadfactor*, the fewer requests the Tomcat instance will be given. If a Tomcat instance is not responding, *mod\_proxy\_balancer* marks that instance as unavailable and fails over to the next instance in the list.

Next, configure and run the Tomcat instances on the Tomcat server computers. Set up your Java environment on tc1 and tc2:

*[root@tomcat: ~]# JAVA\_HOME=/usr/java/jdk1.7.51\_02*

*[root@tomcat: ~]# export JAVA\_HOME*

*[root@tomcat: ~]# PATH=$JAVA\_HOME/bin:$PATH*

*[root@tomcat: ~]# export PATH*

*[root@tomcat: ~]# java –version*

java version "1.7.51\_02"

Java(TM) SE Runtime Environment (build 1.7.51\_02-b06)

Java HotSpot(TM) 64-Bit Server VM (build 1.7.51\_02-b06, mixed mode)

Make sure that CATALINA\_HOME is set on tc1 and tc2:

*[root@tomcat: ~]# CATALINA\_HOME=/usr/local/apache-tomcat-7.0.14*

*[root@tomcat: ~]# export CATALINA\_HOME*

*[root@tomcat: ~]# cd $CATALINA\_HOME*

Then, on each of the Tomcat instance machines, configure the *CATALINA\_HOME/ conf/server.xml* file so that the Engine’s *jmvRoute* is set to the same string you set the Tomcat instance’s *tomcatId* to in the *workers2.properties* file:

*<Engine name="Catalina" defaultHost="localhost" jvmRoute="tomcat1">*

Set the second Tomcat instance’s jvmRoute to "tomcat2", etc. Each Tomcat instance’s *jvmRoute* value must be unique.

Also, in the same file, make sure that the Connector you’re using is configuredproperly for being used through mod\_proxy. See the module [12](#_bookmark814) for the details of the necessary configuration changes.

To test that the request distribution is indeed working, we’ll add some test content. In each Tomcat instance’s webapps/ROOT/ directory, do the following:

*[root@tomcat: ~]# cd $CATALINA\_HOME/webapps/examples*

*[root@tomcat: ~]# echo 'Tomcat1' > instance.txt*

Do the same in the second Tomcat’s webapps/ROOT/ directory, labeling it as

Tomcat2:

*[root@tomcat: ~]# cd $CATALINA\_HOME/webapps/examples*

*[root@tomcat: ~]# echo 'Tomcat2' > instance.txt*

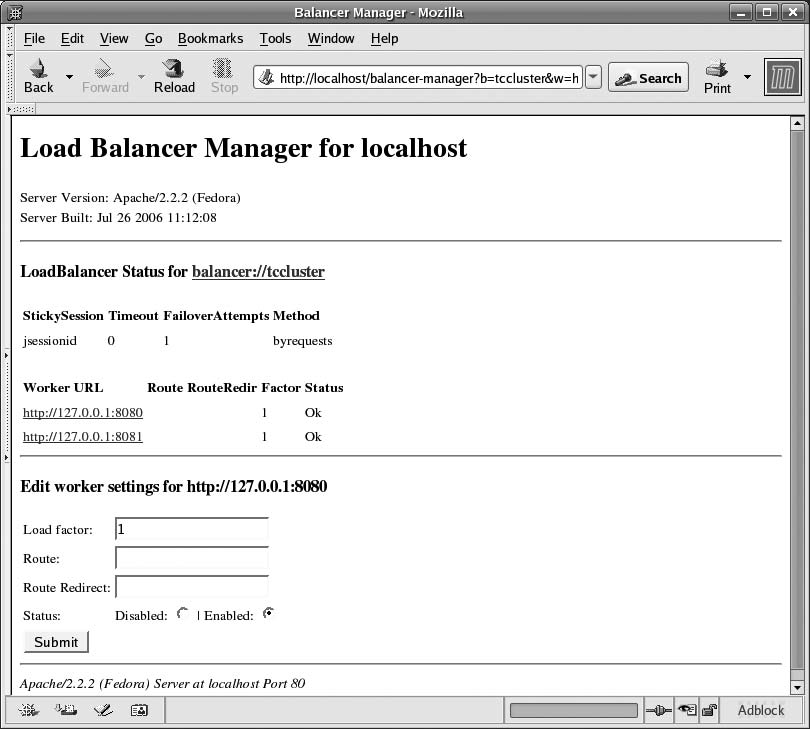
Then, start up each of the two Tomcat instances:

*[root@tomcat: ~]# cd $CATALINA\_HOME*

*[root@tomcat: ~]# bin/catalina.sh start*

Once it’s all running, access the Apache httpd instance on the apache1 machine, and request the instance.txt page by loading the URL <http://apache1/examples/instance.txt> in your browser. The first request will likely be slow because Tomcat initializes everything on the first request. The page will display either Tomcat1 or Tomcat2, depending on which Tomcat instance mod\_proxy\_balancer sent you to. Reloads of the same URL should send you back to the same instance each time, proving that mod\_proxy\_balancer is performing session affinity load balancing.

Try accessing mod\_proxy\_balancer’s /balancer-manager page by loading the URL <http://apache1/balancer-manager> in your browser. It shows information about mod\_ proxy\_balancer’s cluster of configured load balanced and proxied backend server instances. [Figure 10-4](#_bookmark1602) shows what ours looks like, running one Apache httpd with mod\_proxy that is load balancing across two Tomcat instances, all running on the same computer.



*Figure 10-4. mod\_proxy\_balancer’s /balancer-manager page displaying cluster of two Tomcats*

If you click on one of the “Worker URL” links, the form at the bottom of the page shows where you may enable or disable a node in the load balance group. The page refers to a node as a “worker”—it performs work in the cluster.

For more details about load balancing with *mod\_proxy\_balancer*, see the Apache httpd mod\_proxy\_balancer documentation page at [http://httpd.apache.org/docs/2.3/mod/ mod\_proxy\_balancer.html](http://httpd.apache.org/docs/2.3/mod/mod_proxy_balancer.html).

**Distributed Java Servlet Containers**

The Java Servlet Specification version 2.2 (long ago) defined and specified the semantics of distributed servlet containers and the servlet specifications 2.3 through 2.5 further clarified them. The specifications define the behavior and leave much of the implementation detail up to the servlet container authors. Part of what they specify is behavior that can only be implemented as part of the core of any servlet container—a distributed-aware facility built into that core. Specification-compliant distributed servlet container functionality can never be implemented without the servlet container core being aware of the distributed servlet container behavior that the Java Servlet Specifications describe.

Tomcat was originally architected as a non distributed servlet container, but since that initial version, work has been done to implement all of the required features to allow it to operate as a distributed servlet container when it is properly configured for that purpose. Even if you ran your webapp in a specification-compliant distributed servlet container, your web applications may not be able to take advantage of these distributed container features unless your webapps are written to operate as distributed webapps. Here is how the specification describes what a distributable web application is:

A web application that is written so that it can be deployed in a web container distributed across multiple Java virtual machines running on the same host or different hosts. The deployment descriptor for such an application uses the distributable element.

Marking the web application as being distributable in the application’s web.xml file means that it will be deployed and run in a special way on a distributed servlet container. Typically, this means that the author of the web application knows how the distributed servlet container will deploy and run the application, as opposed to how it would be deployed and run in a no ndistributed servlet container.

A distributed servlet container will deploy and run one instance of the application per servlet container, with each servlet container and web application in a separate Java virtual machine, and requests will be processed in parallel. Each JVM may be on its own server computer—in Tomcat’s case, the administrator does the deployment, either through the Manager or Admin applications, or through moving WAR files around and restarting the Tomcat instance(s). Additionally, each Tomcat instance runs its own instance of the web application and treats the application instance as though it is the only one running.

**Servlet** **sessions**

Because there are at least a couple of ways to distribute requests to multiple servlet container instances, the Java Servlet Specification chose one request distribution model for web applications that are marked distributable:

Within an application marked as distributable, all requests that are part of a session must handled by one VM at a time.

This means that requests are handled in parallel by any and all server instances in the cluster, but that all requests belonging to the same session from a single client must be processed by the same servlet container instance. Thus, for webapps that are servlet- specification-compliant distributable webapps, the only compliant request distribution method is session affinity request distribution. If your distributable webapp does not use sessions, you must set up a request distribution method that always sends one client’s requests to the same Tomcat instance without using the JSESSIONID cookie to determine the destination. That is, to stay fully specification compliant.

Conversely, this means that if a client makes several concurrent requests for a distributable web application, your cluster must not distribute those requests to different servlet container instances. Specifying this model for specification-compliant distributable web applications makes it easier for everyone because developers don’t need to worry about concurrent servlet Session object modifications that occur across multiple server computers and multiple JVMs. Also, because all requests that belong to one servlet session must be processed by the same servlet container instance, Session object replication is an optional feature of distributed servlet containers. Here’s what the specification says about that:

The Container Provider can ensure scalability and quality of service features like load- balancing andfailover by having the ability to move a session object, and its contents, from any active node of the distributed system to a different node of the system.

Note that the “can” in the above sentence implies that session replication is an optional runtime feature, so it is not mandatory for distributed servlet containers to perform session replication. The specification does say:

The distributed servlet container must support the mechanism necessary for migrating objects that implement *Serializable*.

This means that specification compliant distributed servlet containers must at least implement session replication for all objects in the session that implement *Serializeable*.

With these exceptions, the behavior of a distributed servlet container is the same as a nondistributed servlet container. For web application authors, it’s important to understand that you probably need to treat user state data differently in distributed applications.

**Session affinity**

When you have your cluster set up to examine the HTTP session cookie and *jvmRoute* and send all dynamic content requests from the same session to the same Tomcat instance, you’re using the session affinity request distribution model. That just means that all requests from the same session are served by the same Tomcat instance.

The terms session affinity and sticky sessions are usually used interchangeably.

*mod\_proxy\_balancer* supports Tomcat session affinity. By default, when Apache httpd forwards a request to *mod\_proxy\_balancer, mod\_proxy\_balancer* examines the session cookie and the jvmRoute and forwards the request to the same Tomcat instance that created the session.

For web applications that are marked distributable, this model is the only model that should be used, per the Java Servlet Specification. When all requests belonging to one HTTP session are served by one Tomcat instance, session replication is not necessary for the application to function under normal circumstances. Of course, if the Tomcat instance fails, or the server machine it runs on fails, the servlet session data is lost. Even if there are more Tomcat instances running in the cluster, the session data was never replicated anywhere; as a result, on the next HTTP request (handled by another Tomcat instance), the user will find that her session state data is gone. Session affinity by itself without session replication is a clusterable solution, but it is not very fault tolerant. It is partially fault tolerant in that if one Tomcat machine out of 10 has an operating system fault that stops Tomcat from answering requests, the other 9 are still properly answering requests. But, it is not fault tolerant for the users whose sessions happen to have been on the only Tomcat machine that stopped responding. On their next request, the load balancer will send them to a different server that does not have a copy of their session state, so they lose the data and must start over.

**Replicated sessions**

With replicated sessions, if one Tomcat instance crashes, the session state data is not lost because at least one other Tomcat instance has been sent a copy of that data.

There are many ways that distributed servlet containers may replicate session data. Some servlet session replication implementations replicate all sessions to all servlet container instances in the cluster, whereas other implementations replicate one servlet container instance’s sessions to only one or two other “buddy” servlet container instances in the cluster.

The network protocol over which session data is replicated also varies. Any replication implementation may offer one or more of the following protocol choices:

***TCP Unicast***

This is a reliable protocol, but it generates quite a bit of network traffic overhead. It’s also a one-to-one communication protocol, which requires sending duplicate network packet data to each instance that will receive session data. It’s probably the easiest to set up and run but is the most demanding protocol on network bandwidth resources.

***Unreliable Multicast Datagram***

This protocol has no built-in error correction, delivery guarantee, or delivery ordering, but it’s a one-to-many protocol that can greatly reduce network traffic. Each instance in the multicast group receives everything sent to that multicast group by any group member. Because each Tomcat instance receives all communication traffic, each server machine’s CPU may become busied with listening in on the group’s chatter.

***Reliable Multicast Datagram***

This is the same as the unreliable multicast with an added reliability layer. There is no single industry standard for it; every reliable multicast library implements the algorithm somewhat differently. Implementations can add data to the multicast packets to keep track of delivery ordering, delivery priority, delivery acknowledgments, resend requests, resend replies, and so on. The CPU overhead is higher than for unreliable multicast because of the extra layer of code that handles reliability, and the network utilization is a little higher too because of the extra reliability data in the network packets. But, unlike TCP unicast, this protocol can do one-to-many communications without duplicating packets for each server in the cluster.

Over these protocols, session replicators speak their own higher-level custom application protocol that is all about exchanging session data updates. For instance, one kind of message sent from one Tomcat instance to all other Tomcat instances in the cluster could mean “I’ve created a new empty session numbered 123456,” and all of the instances that receive this message would know to duplicate that session in their JVMs.

**Tomcat Clustering Implementation**

Tomcat 7 has a new clustering implementation compared with that of Tomcat 5.5 and earlier. This module covers the Tomcat 7 clustering implementation. The new configuration enables users to take advantage of plugging in their own message interceptors and to do primary/secondary backup session replication as opposed to the all-to-all session replication that was the only choice in earlier versions of Tomcat.

You may be asking yourself, “What would I need to change in my webapp to make it run as a distributed webapp?” The good news is it is likely the case that you would not have to change anything except for adding the *<distributable/>* element in your webapp’s *WEB-INF/web.xml* file. As long as you use session affinity (sticky session) load balancing, theoretically the webapp does not have to know that any session replication is going on. But, it depends on what your webapp does. The session is transparently replicated across the cluster, but not synchronously. So, all of a single client’s requests have to keep going to the same Tomcat instance so that the session’s state data is always seen in a consistent manner until either the session is invalidated or the Tomcat instance fails. In most cases, the webapp can be used in a non clustered configuration (development) and in a clustered (production) configuration, without any changes to the webapp itself.

Before we dig into the details of Tomcat’s clustering implementation, here are some specific terms that are usedto describe it. These terms may be used in other contexts and have other meanings, but when discussing Tomcat 7’s clustering implementation, this is how they are defined:

***Manager***

A web application session manager. When a webapp is distributed, the session manager implementation must be a replicated session manager, so Tomcat implements a couple of those: DeltaManager and BackupManager.

***Group***

A logical grouping of Tomcat nodes, each member of the group participates in the clustering effort in a particular way. For example, one group could be the session replication group for the ROOT webapp of the [www.example.com](http://www.example.com/) host, while another group could replicate the sessions for the ROOT webapp of the [www.](http://www/) groovywigs.com host. Even though these two groups replicate different things, the same Tomcat nodes could participate in both logical groups.

***Member***

A participating node of the Tomcat cluster. Any Tomcat node may be a member of zero or more groups.

***Channel***

Group communications framework software that includes facilities to send and receive group messages of various types, and it propagates cluster membership join/leave events. All cluster communications pass through this Channel object.

***Sender***

This Channel object sends replication data from the Tomcat node on which the data is being modified to the node(s) that are replicating the data.

***Receiver***

This corresponds to a Sender, receiving all replication data that is sent by the Sender and handing the data off to the proper consumer. Think of the Receiver as an additional network server software component that runs inside the same JVM as Tomcat that receives all replication data. Depending on the implementation class used for the Receiver (it is configurable in server.xml), the Receiver may implement a thread pool for better performance and scalability.

***Interceptor***

A software component that intercepts message communications between the channel and the IO layer. Interceptors may act on the data in any way a developer programs them to, including modifying the data and sending it through, dropping the data, sending additional data, storing the data, and so on.

***Transport***

An implementation of pluggable communications software that transmits and receives cluster messages via a specific network protocol. Tomcat 7 includes two implementations of transports: nonblocking Java (dubbed the “NIO” implementation) and blocking Java IO (dubbed “BIO,” although the same kind of blocking Java IO implementation as a Connector is referred to as “JIO”).

***Heartbeat***

If you use the default multicast node discovery and group notification implementation, the nodes each send out “heartbeat” messages to all other nodes that are listening once every half a second (the frequency is configurable). Other nodes that are listening via multicast will receive the heartbeat messages and can discover the existence of the other nodes because of these messages. Each node keeps track of which other nodes it has heard from and keeps listening for the heartbeat messages. If a node’s heartbeat messages aren’t heard anymore, the node is considered nonfunctional, and it is removed from the cluster of participating members until it is rediscovered (the same way it was originally discovered). The heartbeat messages may also contain data and are used for carrying small messages to the other nodes in the cluster.

**Features**

Tomcat 7’s clustering code implements many great features. Here is a list of features that are included:

***Cluster group membership***

Each Tomcat node can be configured to be a member of one or more cluster groups. Once the node joins the group at runtime, the node can send/receive messages to other nodes in the group. The Tomcat “tribes” framework implements group registration, join/leave group message propagation, and node failure detection.

***Group message interceptors***

Tomcat’s code implements several group message interceptor classes that can customize group message transmission. This includes interceptors to filter for a domain, fragment/reassemble large messages, .gzip compress the communications, order the messages, detect TCP failures, and log message throughput summary statistics.

***Pluggable session replication schemes***

Tomcat includes code for more than one cluster message replication scheme, including an all-to-all node replication scheme that sends only the diffs of the session data that changed (DeltaManager) and a primary/secondary backup scheme—where one Tomcat node is the session affinity primary node that gets all of the session’s requests and another node is the replicated backup (secondary) node that receives diffs of the session data changes (BackupManager).

***Replicated context attributes***

Servlets andJSPs of a web application may set attribute values on the webapp’s context (“application scope”) anduse these values as webapp state data. But, if the webapp is distributed and running on multiple nodes, the context state may need to be replicated for the webapp to still operate properly in a distributed environment without first redesigning the webapp. The servlet specification frowns upon replicating context attributes:

SRV.3.4.1 Context Attributes in a Distributed Container. Context attributes are local to the JVM in which they were created. This prevents *ServletContext* attributes from being a shared memory store in a distributed container. When information needs to be shared between servlets running in a distributed environment, the information should be placed into a session, stored in a database, or set in an Enterprise JavaBeans™ component.

But if the webapp is already written to use context attributes as an application-wide temporary state storage mechanism, the context attributes must be replicated, or the webapp would need to be redesigned to run properly as a distributed webapp. Tomcat implements replicated context attributes via the *ReplicatedContext* class in an attempt to save you from having to redesign the webapp before you can run it in a Tomcat cluster. But, the specification is right such that the data should be stored elsewhere (in a database or in the user’s session), so this feature should be used only as a temporary workaround.

***Cluster-wide single sign-on authentication***

This is similar to single node single sign-on, but extends it across the cluster. Once a client successfully authenticates with one webapp in one Tomcat node, the client is also authenticated on all other nodes for all webapps in the same Host.

***Pluggable cluster components***

In the standard Tomcat architecture tradition, all of Tomcat’s clustering code components that get configured in server.xml are pluggable in that you may write your own implementation and plug it in via a configuration change in server.xml. Pluggable clustering components include (but are not limited to) implementations for the Channel, ClusterListener, Interceptor, Manager, Membership, Receiver, Sender, Transport, and Valve components.

In Tomcat 7.0, the Cluster group webapp (re)deployment/undeployment feature is broken (the FarmWarDeployer), and as of this writing, it is a future “To Do” to write something new to implement the feature. This is meant to allow (re)deploying a distributable (cluster-aware) web application once for the entire cluster of Tomcat nodes. The webapp would be distributed to all nodes in the cluster and could be started automatically on all nodes. Undeployment also would work cluster-wide. This was previously implemented in the *FarmWarDeployer* class and its associated classes, however, in Tomcat 7.0 it does not work. The “farm” part of the name comes

from the traditional term “cluster farm.” But, see this page for an Ant build file that can deploy a webapp to all nodes in the cluster: [*http://marc.info/?l=tomcat-user&m=118062794431088&w=2*](http://marc.info/?l=tomcat-user&amp;m=118062794431088&amp;w=2).

**Configuring and Testing** **IP** **Multicast**

You cannot assume that multicast will just work. Not all operating systems support it, nor do some network devices. It will likely work well on popular UN\*X-like operating systems, however.

On Solaris, it’s likely to already be set up and working in a stock installation:

*# ifconfig -a*

*lo0: flags=1000849<UP,LOOPBACK,RUNNING,MULTICAST,IPv4> mtu 8232 index 1*

*inet 127.0.0.1 netmask ff000000*

*hme0: flags=1000843<UP,BROADCAST,RUNNING,MULTICAST,IPv4> mtu 1500 index 2*

*inet 10.1.0.1 netmask ffff0000 broadcast 10.1.255.255*

The hme0 Ethernet interface shows MULTICAST on a computer we tested, and it just worked.

Getting IP multicast working on Linux is a little tougher, as it may require a kernel recompile (usually it does not). To see if your kernel supports multicast, try this:

*# cat /proc/net/dev\_mcast*

*9 eth1 1 0 01005e000001*

If the indicated file doesn’t exist, you will likely need to recompile your kernel to support multicast. It’s one kernel option: CONFIG\_IP\_MULTICAST. Turn the option on, recompile, and reboot.

If your kernel already supports multicast, you need to make sure that multicast is enabled on your network device. Regardless of whether you’re doing multicast over eth0, eth1, or local loopback (lo), you must use ifconfig to enable multicast on that device. To find out if multicast is enabled, just use ifconfig to examine the device’s settings, like this:

*# ifconfig -a*

*eth0 Link encap:Ethernet HWaddr 00:10:A4:8E:65:D6*

*inet addr:10.1.0.1 Bcast:10.1.255.255 Mask:255.255.0.0 UP BROADCAST MTU:1500 Metric:1*

*RX packets:338825 errors:0 dropped:0 overruns:0 frame:0*

*TX packets:132580 errors:0 dropped:0 overruns:38 carrier:0 collisions:0 txqueuelen:100*

*Interrupt:11*

*lo Link encap:Local Loopback*

*inet addr:127.0.0.1 Mask:255.0.0.0*

*UP LOOPBACK RUNNING MTU:16436 Metric:1*

*RX packets:27174 errors:0 dropped:0 overruns:0 frame:0 TX packets:27174 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0*

Looking at eth0, we don’t see MULTICAST listed, so we use ifconfig to enable it:

*# ifconfig eth0 multicast*

*# ifconfig -a*

*eth0 Link* *encap:Ethernet HWaddr 00:10:A4:8E:65:D6*

*inet addr:10.1.0.1 Bcast:10.1.255.255 Mask:255.255.0.0 UP BROADCAST MULTICAST MTU:1500 Metric:1*

*RX packets:338825 errors:0 dropped:0 overruns:0 frame:0*

*TX packets:132580 errors:0 dropped:0 overruns:38 carrier:0 collisions:0 txqueuelen:100*

*Interrupt:11*

*lo Link encap:Local Loopback*

*inet addr:127.0.0.1 Mask:255.0.0.0*

*UP LOOPBACK RUNNING MTU:16436 Metric:1*

*RX packets:27224 errors:0 dropped:0 overruns:0 frame:0 TX packets:27224 errors:0 dropped:0 overruns:0 carrier:0 collisions:0 txqueuelen:0*

Now that multicast is enabled, add the IP route for the multicast class D network. On the multicast-enabled device that you want to handle the multicast traffic, add a route like this:

*# route add -net 224.0.0.0 netmask 240.0.0.0 dev eth0*

Feel free to change the eth0 on the end to the device of your choice, but if you change anything else in this command line, multicast probably won’t work.

If route complains about the *netmask*, you’re probably not adding the route with the *-net* option.

Next, you shouldtest multicasting. [Example 13-1](#_bookmark1649) is a Java program that you can use to test IP multicast on a single machine or between two machines on a LAN.

*Example 13-1. MulticastNode.java*

import java.net.DatagramPacket; import java.net.InetAddress; import java.net.MulticastSocket;

/\*\*

* MulticastNode is a very simple program to test multicast. It starts
* up and joins the multicast group 228.0.0.4 on port 45564 (this is the
* default address and port of Tomcat 6's Cluster group communications).
* This program uses the first argument as a message to send into the
* multicast group, and then spends the remainder of its time listening
* for messages from other nodes and printing those messages to standard
* output.

\*/

public class MulticastNode {

*Example 10-1. MulticastNode.java (continued)*

InetAddress group = null; MulticastSocket s = null;

/\*\*

* Pass this program a string argument that it should send to the
* multicast group.

\*/

public static void main(String[] args) { if (args.length > 0) {

System.out.println("Sending message: " + args[0]);

// Start up this MulticastNode MulticastNode node = new MulticastNode( );

// Send the message node.send(args[0]);

// Listen in on the multicast group, and print all messages node.receive( );

} else {

System.out.println("Need an argument string to send."); System.exit(1);

}

}

/\*\*

\* Construct a MulticastNode on group 228.0.0.4 and port 45564.

\*/

public MulticastNode( ) { try {

group = InetAddress.getByName("228.0.0.4"); s = new MulticastSocket(45564); s.joinGroup(group);

} catch (Exception e) { e.printStackTrace( );

}

}

*Example 13-1. MulticastNode.java (continued)*

/\*\*

* Send a string message to the multicast group for all to see.

\*

* @param msg the message string to send to the multicast group.

\*/

public void send(String msg) { try {

DatagramPacket hi = new DatagramPacket( msg.getBytes(), msg.length( ), group, 45564);

s.send(hi);

} catch (Exception e) { e.printStackTrace( );

}

}

/\*\*

* Loop forever, listening to the multicast group for messages sent
* from other nodes as DatagramPackets. When one comes in, print it
* to standard output, then go back to listening again.

\*/

public void receive( ) { byte[] buf;

// Loop forever while (true) {

try {

buf = new byte[1000];

DatagramPacket recv = new DatagramPacket(buf, buf.length); s.receive(recv);

System.out.println("Received: " + new String(buf));

} catch (Exception e) { e.printStackTrace( );

}

}

}

}

Compile this class:

*[root@tomcat: ~]#* *javac MulticastNode.java*

Then, run the first node:

$ java MulticastNode NodeOne Sending message: NodeOne Received: NodeOne

The Received: NodeOne message indicates that NodeOne is receiving its own multicast group join message. It will receive everything sent to the multicast group, including everything it transmits to the group.

In another shell, run the second node:

*[root@tomcat: ~]#* java MulticastNode NodeTwo Sending message: NodeTwo Received: NodeTwo

Then, look back at the output of NodeOne;

it shouldlook like this once NodeTwo joins

NodeOne’s multicast group:

Sending message: NodeOne

Received: NodeOne

Received: NodeTwo

This means that NodeOne received NodeTwo’s join message via IP multicast! If that works, you

should be able to stop NodeTwo (with a Ctrl-C) and restart it and see another Received: NodeTwo message in NodeOne’s output. If all that works, your OS’s multicast is ready to use.

**Configuring All-to-All Replication**

This is the most common configuration for Tomcat session replication—two or more Tomcat nodes, and each node sends replication messages to each of the other nodes in the cluster. With this configuration, you would still have fault tolerance if all of your servers crashed except for a single server because all sessions would failover to the one remaining node.

The disadvantage to this configuration is that because all replication messages go out to all nodes, all nodes incur some CPU overhead for session changes in all other nodes, so the network and CPUs become extra busy. This configuration scales up to a certain number of nodes only and because the more nodes you have, the more rep- lication messages would need to be handled by each CPU. If you have only a small number of Tomcat nodes in your cluster, this replication configuration should work fine. You’ll know you have too many nodes for this configuration when the CPUs get busy enough that Tomcat serves requests significantly slower. Benchmarking some requests with different sized groups will show you how many is too many for your particular set of webapps (it really depends on how often your webapp makes session data changes). If you find that you have too many nodes in your cluster for all-

to-all replication, you can either segment your network such that half of your nodes are in one group and half are in another group, or you can use the primary/backup replication configuration described later in the “Configuring Primary/Backup Replication” section.

Here is what you will need before you begin configuring Tomcat to do all-to-all replication:

* Your webapp must run on a Java JDK version 1.5.0 or higher. (You must already do this if you are running it on Tomcat version 6.0 or higher.)
* Your webapp must only add objects to users’ sessions that properly implement the *java.io.Serializeable* interface. If any objects added to the session are not *serializeable*, session replication attempts will not work.
* One or more server machines capable of running two or more Tomcat instances. For instance, you can run two Tomcat instances on a single machine to test Tomcat session replication and, optionally, context attribute replication. Or, you can set up two or more server machines, each machine running a single instance of Tomcat. If your instances are on separate machines, just make sure that the two machines are communicating with each other properly before configuring and testing Tomcat clustering. For example, if you are going to use the multicast group membership *autodiscovery*, make sure that multicast communication is configured and working between all of the server machines first.
* Each of your Tomcat instances must set a unique jvmRoute value on their
* *<Engine>* element in server.xml. This value gets appended to the end of the session cookie, which enables the load balancer to know which Tomcat node to send subsequent requests to.
* Session affinity *loadbalancing*; you must first set up your loadbalancer in front of your Tomcat nodes, and it must be performing session affinity (sticky session) load balancing. One completely free load balancer that you could use for this is Apache httpd and *mod\_proxy\_balancer*, as described earlier in this chapter. It will function properly for distributing requests to the Tomcat nodes. But, as our benchmark in Module 10 shows, putting Apache httpd in front of Tomcat on the request chain can slow Tomcat down significantly. (Other options for load balancing are listed earlier in this chapter.) Hardware loadbalancers are known to offer good performance, but there are also software loadbalancers that work about as well when running on a fast server machine. Whichever load balancer you choose must switch machines based on the JSESSIONID HTTP cookie value. This is also known as “cookie switching.” The value of JSESSIONID will look something like A1CA147ACB78DC986F38A337BB950569.tc8, where the ending tc8 means that the next request is meant to go to the node whose <Engine>’s jvmRoute attribute is set to tc8. If the tc8 node is down or not in the rotation for whatever reason, the load balancer should pick the next best machine, which can be random or based on load averages.

When load balancing is all set up and working, it must be transparent to the client in that the URLs never change from one Tomcat node to another because the cookie domain would end up being different if that is not the case. If the cookie domains are different for each node, making a request to a different node means that the session cookie from the original node would not be used, and a new session cookie is created on the second node—not what you want if you are doing session replication! Without this kind of load balancing, session replication will not work.

* You must synchronize time across all of the server machines that participate in the same cluster group. Some features of Tomcat’s clustering messaging code are time dependent, and a difference in clock time even as small as a second or two could make it malfunction. We highly suggest using Network Time Protocol (NTP) to set your servers’ clocks so that they are properly synchronized.
* If you are going to use the multicast cluster node autodiscovery, you must make sure that multicast works between the computers running each of the Tomcat nodes. If you cannot use multicast or do not wish to use multicast, you must statically configure the cluster group members.
* Now that all of the prerequisites are out of the way, we can configure clustering. First, configure your webapp to be a distributed webapp. Each webapp that you wish to run as a distributed webapp must have the *<distributable/>* element in its *WEB- INF/web.xml* file. This is the servlet specification compliant way to tell the servlet container that the webapp is designed to be able to run in a distributed servlet con- tainer with session replication enabled.

Each *<Context>* for a distributable webapp must have the *distributable="true"* attribute setting. This tells Tomcat that not only is the webapp distributable but that you are directing Tomcat to run the webapp as a distributed webapp (as opposed to the default which is *nondistributed*, *nonreplicated*).

You must have both the *distributable="true"* attribute set on the webapp’s Context and the *<distributable/>* element in the webapp’s web.xml for session clustering to work.

Next, we’ll configure the Tomcat nodes to cluster together as a group. In the first Tomcat’s server.xml file, we’ll add a *<Cluster>* element andsome subelements under Tomcat’s <Engine> element, like this:

*<Engine name="Catalina" defaultHost="*[*www.example.com*](http://www.example.com/)*" jvmRoute="tc1">*

*<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster" channelSendOptions="8">*

*<Manager className="org.apache.catalina.ha.session.DeltaManager" expireSessionsOnShutdown="false" notifyListenersOnReplication="true"/>*

*<Channel className="org.apache.catalina.tribes.group.GroupChannel">*

*<Membership className="org.apache.catalina.tribes.membership.McastService" address="228.0.0.4"*

*port="45564" frequency="500" dropTime="3000"/>*

*<Sender className="org.apache.catalina.tribes.transport.*

*ReplicationTransmitter">*

*<Transport className="org.apache.catalina.tribes.transport.nio.*

*PooledParallelSender"/>*

*</Sender>*

*<Receiver className="org.apache.catalina.tribes.transport.nio.NioReceiver" address="auto"*

*port="4000" autoBind="100" selectorTimeout="5000" maxThreads="6"/>*

*<Interceptor className="org.apache.catalina.tribes.group.interceptors.*

*TcpFailureDetector"/>*

*<Interceptor className="org.apache.catalina.tribes.group.interceptors.*

*MessageDispatch15Interceptor"/>*

*</Channel>*

*<Valve className="org.apache.catalina.ha.tcp.ReplicationValve" filter=""/>*

*<Valve className="org.apache.catalina.ha.session.JvmRouteBinderValve"/>*

*<ClusterListener className="org.apache.catalina.ha.session.*

*JvmRouteSessionIDBinderListener"/>*

*<ClusterListener className="org.apache.catalina.ha.session.*

*ClusterSessionListener"/>*

*</Cluster>*

Under your Tomcat’s <Engine> line in server.xml file, add these configuration lines in each Tomcat node that you want to be part of the cluster group. Make sure, though, that the *<Engine>* element’s jvmRoute attribute is set to a different value in each of your Tomcat nodes.

What does the configuration mean? Each element and its nested elements serve a small function that together make up the necessary clustering features. Each element’s function is explained in Module 4, but below is a terse description of what the above configuration lines do:

**<Cluster>**

Serves as the container element for all of the clustering configuration tags.

**<Manager>**

Specifies a clustered session manager implementation for a node to use.

**<Channel>**

Configures the group communication “channel” implementation used by the cluster.

**<Membership>**

Configures how the cluster members (nodes) find each other and how they keep track of which nodes are up and running.

**<Receiver>**

Specifies and configures the implementation of the code that receives cluster replication messages. The Receiver receives cluster messages that were sent by another node’s Sender.

**<Sender>**

Specifies and configures the implementation of the code that sends replication messages out to other cluster members (nodes).

**<Transport>**

Specifies the pluggable transport implementation that will be used by a Sender

(but not a Receiver).

**<Interceptor>**

Code modules that can act on or modify messages leaving the Sender, or entering the Receiver, or both.

**<Valve>**

Regular Tomcat Valve implementations that can modify requests and/or responses.

**<ClusterListener>**

Configures the intended recipient code modules of cluster messages, such as session replication messages. ClusterListeners are similar to Interceptors, but they’re meant to be the final destination for certain types of cluster messages, whereas Interceptors are listening into the communication between the sender and the Receiver’s ClusterListener, and may intervene.

With the Cluster configuration shown above, you should be able to run each Tomcat node and each node should automatically discover each other node in the cluster via multicast. Then, once the nodes are aware of each other, they can begin replicating session data to each other as long as the same version of the same webapp is deployed on all nodes that are participating in the webapp’s session replication.

Leave all of the numbers of the Cluster configuration the same on all nodes, including the Membership address and port attribute values. The code is smart enough to figure out how to use the network to communicate without interfering with the networking of another Tomcat node, even when more than one node is running on the same physical computer.

In the case where you’re testing Tomcat clustering on a single computer, just make sure that the *<Server>* element’s shutdown port number and the *<Connector>* address

or port numbers are different values for each Tomcat JVM, and everything should run smoothly. If you forget to change the shutdown port number or the connector address or port numbers to be unique for each JVM, you will see errors in the logs because two JVMs cannot open server sockets on the same host and port number.

When you start your Tomcat nodes, in the catalina.out log, you should see something like this:

*INFO: Starting Servlet Engine: Apache Tomcat/6.0.14*

*Sep 27, 2008 7:07:39 PM org.apache.catalina.ha.tcp.SimpleTcpCluster start INFO: Cluster is about to start*

*Sep 27, 2008 7:07:39 PM org.apache.catalina.tribes.transport.ReceiverBase bind INFO: Receiver Server Socket bound to:www.example.com:4000*

*Sep 27, 2008 7:07:39 PM org.apache.catalina.tribes.membership.McastServiceImpl setupSocket*

*INFO: Setting cluster mcast soTimeout to 500*

*Sep 27, 2008 7:07:39 PM org.apache.catalina.tribes.membership.McastServiceImpl waitForMembers*

*INFO: Sleeping for 1000 milliseconds to establish cluster membership, start level:4 Sep 27, 2008 7:07:40 PM org.apache.catalina.tribes.membership.McastServiceImpl waitForMembers*

*INFO: Done sleeping, membership established, start level:4*

*Sep 27, 2008 7:07:40 PM org.apache.catalina.tribes.membership.McastServiceImpl waitForMembers*

*INFO: Sleeping for 1000 milliseconds to establish cluster membership, start level:8 Sep 27, 2008 7:07:41 PM org.apache.catalina.tribes.membership.McastServiceImpl waitForMembers*

*INFO: Done sleeping, membership established, start level:8*

*Sep 27, 2008 7:07:42 PM org.apache.catalina.ha.session.DeltaManager start INFO: Register manager /examples to cluster element Engine with name Catalina Sep 27, 2008 7:07:42 PM org.apache.catalina.ha.session.DeltaManager start INFO: Starting clustering manager at /examples*

*Sep 27, 2008 7:07:42 PM org.apache.catalina.ha.session.DeltaManager getAllClusterSessions*

*INFO: Manager [www.example.com#/examples]: skipping state transfer. No members active in cluster group.*

*Sep 27, 2008 7:07:42 PM org.apache.catalina.ha.session.JvmRouteBinderValve start INFO: JvmRouteBinderValve started*

*Sep 27, 2008 7:07:42 PM org.apache.coyote.http11.Http11Protocol start INFO: Starting Coyote HTTP/1.1 on http-8080*

*Sep 27, 2008 7:07:42 PM org.apache.jk.common.ChannelSocket init INFO: JK: ajp13 listening on /0.0.0.0:8009*

*Sep 27, 2008 7:07:42 PM org.apache.jk.server.JkMain start INFO: Jk running ID=0 time=0/26 config=null*

*Sep 27, 2008 7:07:42 PM org.apache.catalina.startup.Catalina start INFO: Server startup in 4128 ms*

*Sep 27, 2008 7:09:45 PM org.apache.catalina.tribes.io.BufferPool getBufferPool INFO: Created a buffer pool with max size:104857600 bytes of type:org.apache. catalina.tribes.io.BufferPool15Impl*

*Sep 27, 2008 7:09:46 PM org.apache.catalina.ha.tcp.SimpleTcpCluster memberAdded*

*INFO: Replication member added:org.apache.catalina.tribes.membership.MemberImpl[tcp:/*

*/tc2.example.com:4001,tc2.example.com,4001, alive=1208,id={56 -55 94 23 26 -33 72 -66*

*-101 -47 109 5 -122 89 51 68 }, payload={}, command={}, domain={}, ]*

This shows that the Tomcat node tc1 started up successfully and then automatically discovered another node named tc2 and added tc2 as a member of the cluster group. At that point, the two nodes are ready to replicate session data via TCP. They have exchanged TCP host and port information and have connected to each other via TCP for replication communications.

If you do not see the *SimpleTcpCluster* member Added message in your *catalina.out* logfile, you should recheck your server.xml files to make sure that you have the configuration set correctly and also retest multicast communications between the computers where the Tomcat nodes are running. Once you have it configured correctly, its node auto discovery will work and you will see these messages in the log with stock log settings.

**Testing Session Replication**

To be sure that your cluster is doing what you want it to do, you should test your configuration. Some things you should try to do include changing a session and watching the logfiles on other nodes to make sure that they are receiving replicated session data and testing various failures to make sure that sessions are being moved from one node to another properly with no session data loss.

To test your clustering configuration, you need to deploy a distributed webapp that you can use to make changes to a session in a carefully controlled manner. Tomcat comes with a webapp that contains servlet examples, one of which contains a servlet example that allows the user to type in a session attribute name and value (try it at <http://yourhost:8080/examples/servlets/servlet> /SessionExample). [Figure 13-5](#_bookmark1669) shows the *SessionExample* page as served by our tc1 node.

Because the example webapp includes this servlet, it is an ideal candidate for testing your Tomcat cluster configuration; it just needs to be promoted to being a distributed webapp. To do this, first edit the web.xml file for it andaddthe <distributable/> tag. The web.xml file resides in *CATALINA\_HOME/webapps/examples/WEB-INF/web.xml*. Add a line that only contains the *<distributable/>* tag like this:

[*<web-app xmlns="http://java.sun.com/xml/ns/j2ee"*](http://java.sun.com/xml/ns/j2ee)[*xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"*](http://www.w3.org/2001/XMLSchema-instance)[*xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee*](http://java.sun.com/xml/ns/j2ee)[*http://java.sun.com/xml/ns/*](http://java.sun.com/xml/ns/)

*j2ee/web-app\_2\_5.xsd" version="2.5">*

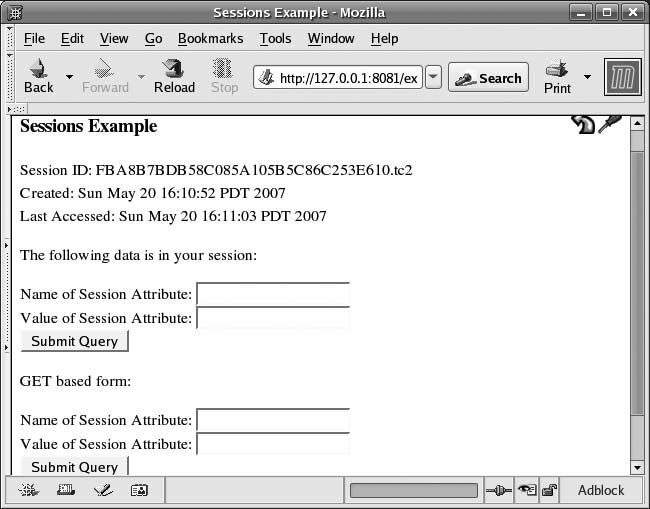
*<description>*

*Servlet and JSP Examples.*

*</description>*

*<display-name>Servlet and JSP Examples</display-name>*

*<distributable/>*



*Figure 13-5. Viewing* *session data via an example servlet*

You’ll also need to add distributed="true" to the webapp’s <Context> element. If there is no <Context> element declared for it anywhere, you will need to create one. You can create one by making a new CATALINA\_HOME/conf/[EngineName]/ [HostName]/examples.xml context XML fragment file or by adding one to server.xml. (Consult [Chapter 3](#_bookmark487) if you need details on how to do this.) Make these changes on all of your Tomcat nodes (you must have two or more, of course). Then, restart them if nec- essary for the deployment change to take effect. At that point, you have a distributed webapp with clusteredsessions, andif you use the servlet to adda session attribute, the attribute and its value should be replicated to the other node(s) in the cluster.

If your webapp’s hostname was [www.example.com,](http://www.example.com/) both Tomcat instances should have this in their [www.example.com.](http://www.example.com/)<date>.logfile after adding a session attribute named 'test' with a value of '1':

*[root@tomcat: ~]#*  *tail* [*www.example.com.2014-09-27.log*](http://www.example.com.2014-09-27.log)

*Sep 27, 2014 5:39:38 PM org.apache.catalina.core.ApplicationContext log INFO: ContextListener: contextInitialized( )*

*Sep 27, 2014 5:39:38 PM org.apache.catalina.core.ApplicationContext log INFO: SessionListener: contextInitialized( )*

*Sep 27, 2014 5:40:07 PM org.apache.catalina.core.ApplicationContext log*

*INFO: SessionListener: sessionCreated('F4B2D7191C1F335FFAAC93DA461CA95F.tc1')*

*Sep 27, 2014 5:40:19 PM* *org.apache.catalina.core.ApplicationContext log*

*INFO: SessionListener: attributeAdded('F4B2D7191C1F335FFAAC93DA461CA95F.tc1', 'test', '1')*

When more replication changes are made to the session, including modifications to existing session attributes, you should see additional log lines like these:

*Sep 27, 2014 5:43:58 PM org.apache.catalina.core.ApplicationContext log*

*INFO: SessionListener: attributeReplaced('29EC385B502CB098FE7B9C9DC22B947B.tc2', 'foo', '2')*

*Sep 27, 2014 5:43:46 PM org.apache.catalina.core.ApplicationContext log*

*INFO: SessionListener: attributeReplaced('29EC385B502CB098FE7B9C9DC22B947B.tc2', 'foo', '3')*

If you are not seeing these lines in the logs, you undoubtedly have something misconfigured. You should go back over your configuration settings, and if it still does not work, you should ask for help either on the tomcat-user mailing list or on the #Tomcat IRC channel at *irc.freenode.net*.

If you do see these log lines in your nodes, congratulations! You now have your own Tomcat cluster. You can move on to test node failures.

Next, you should try bringing your primary node down and see whether another node becomes the primary node. With Tomcat’s clustering implementation, there is always one node that is the primary, and the load balancer tracks this via the JSESSIONID cookie. When the primary node fails, the load balancer should notice that it is down and route requests to one of the other available nodes. Also, the other nodes in the cluster should notice that the primary node’s heartbeat messages are no longer being sent, and one of the remaining nodes should automatically take over as primary for the session when a new request arrives at any of the remaining nodes.

If you are using *mod\_proxy\_balancer* as the loadbalancer, it is easy to disable the primary node via the /balancer-manager. On the session servlet example page, you can see (at the top of the page) that the session ID has the JSESSIONID appended to the end. You can identify the primary node this way. Go into the /balancer-manager and temporarily disable the primary node so that requests are no longer routed there. Then, issue that Tomcat a stop command so it stops sending out heartbeat messages to the other nodes (they should quickly interpret this as node failure). Then, from your web browser, make a new request to view the session servlet example page. Your request should get routedto a different Tomcat instance transparently, andon the end of the session ID shown on the page, you should see a different JSESSIONID appended.

Tomcat’s clustering implementation uses a *LazyReplicatedMap* class that has an algo- rithm to track what to do with sessions when nodes fail. Here is a quote from one of the Tomcat committers about how it works:

The way the LazyReplicatedMap works is as follows:

1. Backup node fails ➝ primary node chooses a new backup node
2. Primary node fails ➝ since Tomcat doesn’t know which node the user will come to their next http request, nothing is done. When the user makes a request, and the session manager says *LazyMap.getSession*(id) and that session is not yet on the server, the *lazymap* will request the session from the backup server, load it up, and set this node as primary. That is why it is called lazy, cause it won’t loadthe session until it is actually needed, and because it doesn’t know what node will become primary, this is decided by the load balancer.—Filip Hanik

As you can see, a backup (replicated) node may fail as well as the primary node, and in either case Tomcat can recover and continue on, as long as there are remaining nodes in the cluster to use.

### **Configuring Static Membership**

You may optionally statically configure the members of your Tomcat cluster instead of using the multicast autodiscovery method. If you have a small cluster with machines that have unchanging hostnames or IP addresses, or you do not want to configure your network for multicast, you may use the *StaticMembershipInterceptor* to specify the list of Members in your server.xml files on each node.

Here is an example of statically configuring a cluster of two nodes. The nodes have IP addresses of 10.1.0.100 and 10.1.0.101:

*<Interceptor className="org.apache.catalina.tribes.group.interceptors. TcpPingInterceptor"*

*staticOnly="true"/>*

*<Interceptor className="org.apache.catalina.tribes.group.interceptors. TcpFailureDetector"/>*

*<Interceptor className="org.apache.catalina.tribes.group.interceptors. StaticMembershipInterceptor">*

*<Member className="org.apache.catalina.tribes.membership.StaticMember" port="4000"*

*host="10.1.0.100" uniqueId="{10,1,0,100,0,0,0,0,0,0,0,0,0,0,0,0}"/>*

*<Member className="org.apache.catalina.tribes.membership.StaticMember" port="4000"*

*host="10.1.0.101" uniqueId="{10,1,0,101,0,0,0,0,0,0,0,0,0,0,0,0}"/>*

*</Interceptor>*

Notice that the port numbers are the same for both nodes, which works just fine because these ports are on two different machines. If both Tomcat nodes were running on the same machine, you would need to make sure they each had different port numbers. Also notice that the unique Id must be a unique list of 16 values, each of which is interpreted as a byte. We put the IP address numbers as the first four of these values, which already makes these IDs unique, so we set the remainder of the values to zeroes. You may set these to anything you like as long as the set of values is unique for each Member.

The Interceptors configured just above the *StaticMemberInterceptor* will detect any failed nodes by using TCP connections instead of multicast, including sending out TCP heartbeat connections

### **Configuring Primary/Backup Replication**

If you have a larger Tomcat cluster, and you need to lower your cluster’s replication communications bandwidth utilization, or you want to lower CPU and memory utilization on your nodes, you may alternatively use the primary/backup replication scheme by configuring Tomcat to use the *BackupManager* instead of the *DeltaManager*. It is a simple configuration change to use the *BackupManager* instead:

[*<Engine name="Catalina" defaultHost="www.example.com"*](http://www.example.com/) *jvmRoute="tc1">*

*<Cluster className="org.apache.catalina.ha.tcp.SimpleTcpCluster" channelSendOptions="8">*

*<Manager className="org.apache.catalina.ha.session.BackupManager" expireSessionsOnShutdown="false" notifyListenersOnReplication="true"*

*mapSendOptions="6"/>*

*<Channel className="org.apache.catalina.tribes.group.GroupChannel">*

By configuring Tomcat this way, each session has a primary node and a single backup node where replication messages are sent. *DeltaManager’s* all-to-all replication had one primary for a session, and all of the other nodes in the cluster acted as backup nodes for that session. If a session instead only has one backup node, the other nodes in the cluster do not need to keep a replica of the session, which saves those nodes CPU time and memory space. Because replication messages are transmitted over TCP unicast, this can also save quite a bit of network bandwidth, depending on how often session data is changed and the memory footprint of the session data.

**JDBC Request Distribution and Failover**

Typical relational database configurations have one database server instance running on one server computer. Even if all of the other components of the system are clustered, a single database server instance could crash and cause the entire site running on the cluster to become unusable. So, some sort of clustering must also be done for the database so it is not a single point of failure.

There are relational database servers that support replication but not parallel use and some that support both replication and parallel use.

In the case where the database supports replication but not parallelization, the database instance that is replicated to becomes a secondary server that the cluster could failover to. In this case, the database driver code (commonly a JDBC driver) would need to know how to connect to each database instance and when to failover to a secondary (replicated) server.

In the case where the database supports parallelization, the database driver could load balance across several database server instances and detect failures. Here are some products and projects that might interest you:

***Oracle RAC***

One commercial parallel relational database server implementation is Oracle Corporation’s Oracle 10g Real Application Clusters (RAC). See [http://www.](http://www.oracle.com/database/rac_home.html) [oracle.com/database/rac\_home.html](http://www.oracle.com/database/rac_home.html) for product information.

***Sequoia: Open source JDBC replication and load balancing (formerly C-JDBC)***

This interesting open source project has set out to make JDBC clustering avail- able to the masses for free. Sequoia: a JDBC clustering library. This used to be the C-JDBC project but has changed names and web sites since then. The project’s home page is now [http://sequoia.continuent.org](http://sequoia.continuent.org/).