**Module 10**

**Performance tuning strategies**

Once you have Tomcat up and running, you will likely want to do some performance tuning so that it serves requests more efficiently on your computer. In this chapter, we give you some ideas on performance tuning the underlying Java runtime and the Tomcat server itself.

The art of tuning a server is a complex one. It consists of measuring, understanding, changing, and measuring again. The following are the basic steps in tuning:

1. Decide what needs to be measured.
2. Decide how to measure.
3. Measure.
4. Understand the implications of what you learned.
5. Modify the configuration in ways that are expected to improve the measurements.
6. Measure and compare with previous measurements.
7. Go back to step 4.

Note that, as shown, there is no “exit from loop” clause—perhaps a representative of real life. In practice, you will need to set a threshold below which minor changes are insignificant enough that you can get on with the rest of your life. You can stop adjusting and measuring when you believe you’re close enough to the response times that satisfy your requirements.

To decide what to tune for better performance, you should do something like the following.

Set up your Tomcat on a test computer as it will be in your production environment. Try to use the same hardware, the same OS, the same database, etc. The more similar it is to your production environment, the closer you’ll be to finding the bottle- necks that you’ll have in your production setup.

On a separate machine, install and configure your load generator and the response tester software that you will use for load testing. If you run it on the same machine that Tomcat runs on, you will skew your test results, sometimes badly. Ideally, you should run Tomcat on one computer and the software that tests it on another. If you do not have enough computers to do that, then you have little choice but to run all of the software on one test computer, and testing it that way will still be better than not testing it at all. But, running the load test client and Tomcat on the same computer means that you will see lower response times that are less consistent when you repeat the same test.

Isolate the communication between your load tester computer and the computer you’re running Tomcat on. If you run high-traffic tests, you don’t want to skew the test data by involving network traffic that doesn’t belong in your tests. Also, you don’t want to busy computers that are uninvolved with your tests due to the heavy network traffic that the test will produce. Use a switching hub between your tester machine and your mock production server, or use a hub that has only these two computers connected.

Run some load tests that simulate various types of high-traffic situations that you expect your production server to have. Additionally, you should probably run some tests with higher traffic than you expect your production server to have so that you’ll be better prepared for future expansion.

Look for any unusually slow response times and try to determine which hardware and/or software components are causing the slowness. Usually it’s software, which is good news because you can alleviate some of the slowness by reconfiguring or rewriting software. In extreme cases, however,

you may need more hardware, or newer, faster, and more expensive hardware. Watch the load average of your server machine, and watch the Tomcat log files for error messages.

In this chapter, we show you some of the common Tomcat things to tune, including web server performance, Tomcat request thread pools, JVM performance, DNS lookup configuration, and JSP precompilation. We end the chapter with a word on capacity planning.

**Measuring Web Server Performance**

Measuring web server performance is a daunting task, to which we shall give some attention here and supply pointers to more detailed works. There are far too many variables involved in web server performance to do it full justice here. Most measuring strategies involve a “client” program that pretends to be a browser but, in fact, sends a huge number of requests more or less concurrently and measures the response times.

You’ll need to choose how to performance test and what exactly you’ll test. For example, should the load test client and server software packages run on the same machine? We strongly suggest against doing that. Running the client on the same machine as the server is bound to change and destabilize your results. Is the server machine running anything else at the time of the tests? Should the client and server be connected via a gigabit Ethernet, or 100baseT, or 10baseT? In our experience, if your load test client machine is connected to the server machine via a link slower than a gigabit Ethernet, the network link itself can slow down the test, which changes the results.

Should the client ask for the same page over and over again, mix several different kinds of requests concurrently, or pick randomly from a large lists of pages? This can affect the server’s caching and multithreading performance. What you do here depends on what kind of client load you’re simulating. If you are simulating human users, they would likely request various pages and not one page repeatedly. If you are simulating programmatic HTTP clients, they may request the same page repeatedly, so your test client should probably do the same. Characterize your client traffic, and then have your load test client behave as your actual clients would.

Should the test client send requests regularly or in bursts? For benchmarking, when you want to know how fast your server is capable of completing requests, you should make your test client send requests in rapid succession without pausing between requests. Are you running your server in its final configuration, or is there still some debugging enabled that might cause extraneous overhead? For benchmarks, you should turn off all debugging, and you may also want to turn off some logging. Should the HTTP client request images or just the HTML page that embeds them? That depends on how closely you want to simulate human web traffic. We hope you see the point: there are many different kinds of performance tests you could run, and each will yield different (and probably interesting) results.

**Load-Testing Tools**

The point of most web load measuring tools is to request one or more resource(s) from the web server a certain (large) number of times, and to tell you exactly how long it took from the client’s perspective (or how many times per second the page could be fetched). There are many web load measuring tools available on the Web— see [http://www.softwareqatest.com/ qatweb1.html#LOAD](http://www.softwareqatest.com/%20qatweb1.html#LOAD) for a list of some of them. A few measuring tools of note are the *Apache Benchmark* tool (ab, included with distributions of the Apache httpd web server at [http://httpd.apache.org](http://httpd.apache.org/)), *Siege* (see [http://](http://www.joedog.org/JoeDog/Siege) [www.joedog.org/JoeDog/Siege](http://www.joedog.org/JoeDog/Siege)), and *JMeter* from Apache Jakarta (see [http://jakarta.](http://jakarta.apache.org/jmeter) [apache.org/jmeter](http://jakarta.apache.org/jmeter)).

Of those three load-testing tools, JMeter is the most featureful. It is implemented in pure multiplatform Java, sports a nice graphical user interface that is used for both configuration and load graphing, is very featureful and flexible for web testing and report generation, can be used in a text-only mode, and has detailed online documentation showing how to configure and use it. In our experience, JMeter gave the most reporting options for the test results, is the most portable to different operating systems, and supports the most features. But, for some reason, JMeter was not able to request and complete as many HTTP requests per second as ab and siege did. If you’re not trying to find out how many requests per second our Tomcat can serve, JMeter works well because it probably implements all of the features you’ll need. But, if you are trying to determine the maximum number of requests per second your server can successfully handle, you should instead use ab or siege.

If you are looking for a command-line benchmark tool, ab works wonderfully. It is only a benchmarking tool, so you probably won’t be using it for regression testing. It does not have a graphical user interface, nor can it be given a list of more than one URL to benchmark at a time, but it does exceptionally well at benchmarking one URL and giving sharply accurate and detailed results. On most non-Windows operating systems, ab is preinstalled with Apache httpd, or there is an official Apache httpd package to install that contains ab, making the installation of ab the easiest of all of the web load-testing tools.

Siege is another good command-line (no GUI) web load tester. It does not come pre- installed in most operating systems, but its build and install instructions are straight- forward and about as easy as they can be, and Seige’s code is highly portable C code. Siege supports many different authentication features and can perform benchmark testing, regression testing, and also supports an “Internet” mode that attempts to more closely simulate the load your webapp would get with many real users over the Internet. With other, less featureful tools, there seems to be spotty support for webapp authentication. They support sending cookies, but some may not support receiving them. And, while Tomcat supports several different authorization methods (basic, digest, form, and client-cert), some of these less featureful tools support only HTTP basic authentication. Form-based authentication is testable with any tool that is able to submit the form, which depends on whether the tool supports submitting a POST HTTP request for the login form submission (JMeter, ab, and siege each sup- port sending POST requests like this). Only some of them do. Being able to closely simulate the production user authentication is an important part of performance testing because the authentication itself is often a heavy weight operation and does change the performance characteristics of a web site. Depending on which authentication method you are using in production, you may need to find different tools that support it.

As this book was going to print, a new benchmarking software package became available: Faban ([http://faban.sunsource.net](http://faban.sunsource.net/)). Faban is written in pure Java 1.5+ by Sun Microsystems and is open source under the CDDL license. Faban appears to be focused on nothing but careful benchmarking of servers of various types, including web servers. Faban is carefully written for high performance andtight timing so that any measurements will be as close as possible to the server’s real performance. For instance, the benchmark timing data is collected when no other Faban code is run- ning, and analysis of the data happens only after the benchmark has concluded. For best accuracy, this is the way all benchmarks should be run. Faban also has a very nice configuration and management console in the form of a web application. In order to serve that console webapp, Faban comes with its own integrated Tomcat server! Yes, Tomcat is a part of Faban. Any Java developers interested in both Tom- cat andbenchmarking can readFaban’s documentation andsource code andoption- ally also participate in Faban’s development. If you are a Java developer, and you are looking for the most featureful, long-term benchmarking solution, Faban is proba- bly what you shoulduse. We didnot have enough time to write more about it in this book, but luckily Faban’s web site has excellent documentation.

**ab: The Apache benchmark tool**

The ab tool takes a single URL and requests it repeatedly in as many separate threads as you specify, with a variety of command-line arguments to control the number of times to fetch it, the

maximum thread concurrency, and so on. A couple of nice features include the optional printing of progress reports periodically and the comprehensive report it issues.

[Example 10-1](#_bookmark680) is an example running ab. We instructed it to fetch the URL 100,000 times with a maximum concurrency of 149 threads. We chose these numbers care- fully. The smaller the number of HTTP requests that the test client makes during the benchmark test, the more likely the test client will give less accurate results because during the benchmark the Java VM’s garbage collector pauses make up a higher percentage of the total testing time. The higher the total number of HTTP requests that you run, the less significant the garbage collector pauses become andthe more likely the benchmark results will show how Tomcat performs overall. You should bench- mark by running a minimum of 100,000 HTTP requests. Also, you may configure the test client to spawn as many client threads as you would like, but you will not get helpful results if you set it higher than the maxThreads you set for your Connector in your Tomcat’s conf/server.xml file. By default, it is set to 150. If you set your tester to exceed this number and make more requests in more threads than Tomcat has threads to receive and process them, performance will suffer because some client request threads will always be waiting. It is best to stay just under the number of your Connector’s maxThreads, such as using 149 client threads.

*Example 10-1. Benchmarking with ab*

*$ ab -k -n 100000 -c 149* [*http://tomcathost:8080*](http://tomcathost:8080)

*This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-2.0* [*Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/*](http://www.zeustech.net/)

[*Copyright 1997-2005 The Apache Software Foundation, http://www.apache.org/*](http://www.apache.org/)

*Benchmarking tomcathost (be patient) Completed 10000 requests*

*Completed 20000 requests*

*Completed 30000 requests*

*Completed 40000 requests*

*Completed 50000 requests*

*Completed 60000 requests*

*Completed 70000 requests*

*Completed 80000 requests*

*Completed 90000 requests*

*Finished 100000 requests*

*Server Software: Apache-Coyote/1.1 Server Hostname: tomcathost*

*Server Port: 8080*

*Document Path: /*

*Document Length: 8132 bytes*

*Concurrency Level: 149*

*Time taken for tests: 19.335590 seconds Complete requests: 100000*

*Failed requests: 0*

*Write errors: 0*

*Keep-Alive requests: 79058*

*Total transferred: 830777305 bytes HTML transferred: 813574072 bytes*

*Requests per second: 5171.81 [#/sec] (mean) Time per request: 28.810 [ms] (mean)*

*Time per request: 0.193 [ms] (mean, across all concurrent requests) Transfer rate: 41959.15 [Kbytes/sec] received*

*Connection Times (ms)*

*min mean[+/-sd] median max*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Connect:* | *0* | *1* | *4.0* | *0* | *49* |
| *Processing:* | *2* | *26* | *9.1* | *29* | *62* |
| *Waiting:* | *0* | *12* | *6.0* | *13* | *40* |
| *Total:* | *2* | *28* | *11.4* | *29* | *65* |
|  |  |  |  |  |  |

*Percentage of the requests served within a certain time (ms)*

|  |  |
| --- | --- |
| *50%* | *29* |
| *66%* | *30* |
| *75%* | *31* |
| *80%* | *45* |
| *90%* | *47* |
| *95%* | *48* |
| *98%* | *48* |
| *99%* | *49* |
| *100%* | *65 (longest request)* |

If you leave off the -k in the ab command line, ab will not use keep-alive connections to Tomcat, which is less efficient because it must connect a new TCP socket to Tomcat to make each HTTP request. The result is that fewer requests per second will be handled, and the throughput from Tomcat to the client (ab) will be smaller (see [Example](#_bookmark682) 10-2).

*Example 10-2. Benchmarking with ab with keep-alive connections disabled*

*$ ab -n 100000 -c 149* [*http://tomcathost:8080/*](http://tomcathost:8080/)

*This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-2.0* [*Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/*](http://www.zeustech.net/)

[*Copyright 1997-2005 The Apache Software Foundation, http://www.apache.org/*](http://www.apache.org/)

*Benchmarking tomcathost (be patient) Completed 10000 requests*

*Completed 20000 requests*

*Completed 30000 requests*

*Completed 40000 requests*

*Completed 50000 requests*

*Completed 60000 requests*

*Completed 70000 requests*

*Completed 80000 requests*

*Completed 90000 requests*

*Finished 100000 requests*

*Server Software: Apache-Coyote/1.1 Server Hostname: tomcathost*

*Server Port: 8080*

*Document Path: /*

*Document Length: 8132 bytes*

*Concurrency Level: 149*

*Time taken for tests: 28.201570 seconds Complete requests: 100000*

*Failed requests: 0*

*Write errors: 0*

*Total transferred: 831062400 bytes HTML transferred: 814240896 bytes*

*Requests per second: 3545.90 [#/sec] (mean) Time per request: 42.020 [ms] (mean)*

*Time per request: 0.282 [ms] (mean, across all concurrent requests) Transfer rate: 28777.97 [Kbytes/sec] received*

*Connection Times (ms)*

*min mean[+/-sd] median max*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *Connect:* | *0* | *18* | *11.3* | *19* | *70* |
| *Processing:* | *3* | *22* | *11.3* | *22* | *73* |
| *Waiting:* | *0* | *13* | *8.4* | *14* | *59* |
| *Total:* | *40* | *41* | *2.4* | *41* | *73* |

*Percentage of the requests served within a certain time (ms)*

|  |  |
| --- | --- |
| *50%* | *41* |
| *66%* | *41* |
| *75%* | *42* |
| *80%* | *42* |
| *90%* | *43* |
| *95%* | *44* |

98% 46

99% 55

100% 73 (longest request)

**Siege**

To use siege to perform exactly the same benchmark, the command line is similar, only you must give it the number of requests you want it to make per thread. If you’re trying to benchmark 100,000 HTTP requests, with 149 concurrent clients, you must tell siege that each of the 149 clients needs to make 671 requests (as 671 requests times 149 clients approximately equals 100,000 total requests). Give siege the -b switch, telling siege that you’re running a benchmark test. This makes siege’s client threads not wait between requests, just like ab. By default, siege does wait a configurable amount of time between requests, but in the benchmark mode, it does not wait. [Example 10-3](#_bookmark688) shows the siege command line and the results from the bench- mark test.

*Example 10-3. Benchmarking with siege with keep-alive connections disabled*

$ siege -b -r 671 -c 149 tomcathost:8080

\*\* siege 2.65

\*\* Preparing 149 concurrent users for battle. The server is now under siege.. done.

Transactions: 99979 hits

Availability: 100.00 %

Elapsed time: 46.61 secs

Data transferred: 775.37 MB

Response time: 0.05 secs Transaction rate: 2145.01 trans/sec

Throughput: 16.64 MB/sec

Concurrency: 100.62

Successful transactions: 99979

Failed transactions: 0

Longest transaction: 23.02

Shortest transaction: 0.00

Some interesting things to note about siege’s results are the following:

* The number of transactions per second that were completed by siege is significantly lower than that of ab. (This is with keep-alive connections turned off in both benchmark clients,\* andall of the other settings the same.) The only explanation for this is that siege isn’t as efficient of a client as ab is. And that points out that siege’s benchmark results are not as accurate as those of ab.
* Siege is not able to test with keep-alive connections turnedon—a feature that siege is missing, at least as of this writing. This means that using siege, you cannot perform the highest performance benchmark testing, although siege also implements other types of

testing that ab does not implement, such as regression testing and an “” mode, where it can generate randomized client requests to more closely simulate real web traffic.

* The throughput reported by siege is significantly lower than that reportedby ab, probably due to siege not being able to execute as many requests per second as ab.
* The reported total data transferred with siege is approximately equal to the total data transferred with ab.
* ab completed the benchmark in slightly more than half the time that siege completed it in; however, we do not know how much of that time siege spent between requests in each thread. It might just be that siege’s request loop is not as optimally written to move on to the next request.

For obtaining the best benchmarking results, we recommend you use ab instead of siege. However, for other kinds of testing when you must closely simulate web traffic from human users, ab is not suitable because it offers no feature to configure an amount of time to wait between requests. Siege does offer this feature in the form of waiting a random amount of time between requests. In addition to that, siege can request random URLs from a prechosen list of your choice. Because of this, siege can be used to simulate human user load whereas ab cannot. See the siege manual page (by running “man siege”) for more information about siege’s features.

**Apache Jakarta JMeter**

JMeter can be run in either graphical mode or in text-only mode. You may run JMeter test plans in either mode, but you must create the test plans in graphical mode. The test plans are stored as XML configuration documents. If you need to change only a single numeric or string value in the configuration of a test plan, you can probably change it with a text editor, but it’s a good idea to edit them inside the graphical JMeter application for validity’s sake.

Before trying to run JMeter to run a benchmark test against Tomcat, make sure that you start JMeter’s JVM with enough heap memory so that it doesn’t slow down while it does its own garbage collection in the middle of trying to benchmark. This is especially important if you are doing benchmark testing in graphical mode. In the bin/jmeter startup script, there is a configuration setting for the heap memory size that looks like this:

*# This is the base heap size -- you may increase or decrease it to fit your*

*# system's memory availablity: HEAP="-Xms256m -Xmx256m"*

It will make use of as much heap memory as you can give it; the more it has, the less often it may needto perform garbage collection. If you have enough memory in the machine on which you’re running JMeter, you should change both of the 256 num- bers to something higher, such as 512. It is important to do this first because this set- ting’s default could skew your benchmark test results.

To create a test plan for the benchmark, first run JMeter in graphical mode, like this:

*$ bin/jmeter*

**JMeter’s** screen is laid out as a tree view on the left and a selection details panel on the right. Select something in the tree view and you can see the details of that item in the details panel on the right. To run any tests, you must assemble and configure the proper objects in the tree, and then JMeter can run the test and report the results.

To set up a benchmark test like the one we did above with both ab and siege, do this:

1. In the tree view, right click on the Test Plan tree node and select Add ➝ Thread Group.
2. In the Thread Group details panel, change the Number of Threads (users) to 149, change the

Ramp-Up Period (in seconds) to 0, and the Loop Count to 671.

1. Right click on the Thread Group tree node and select Add ➝ Sampler ➝ HTTP Request.
2. In the HTTP request details panel, change the Web Server settings to point to your Tomcat server and its port number, and change the Path under the HTTP Request settings to the URI in your Tomcat installation that you would like to benchmark. For instance /.
3. Right click on the Thread Group tree node again and select Add ➝ Post Processors
4. In the top pull-down menu, select File ➝ Save Test Plan as and type in the name of the test plan you wish to save. JMeter’s test plan file extension is .jmx, which has an unfortunate similarity to the unrelated Java Management eXtension (JMX).

[Figure 10-1](#_bookmark702) shows the JMeter GUI with the test plan, assembled and ready to run. The tree view is on the left, and the detail panel is on the right.

Once you are done building and saving your test plan, you are ready to run the benchmark. Choose File ➝ Exit from the top pull-down menu to exit from the graphical JMeter application. Then, run JMeter in text-only mode on the command line to perform the benchmark, like this:

*$ bin/jmeter -n -t tc-home-page-benchmark.jmx*

Created the tree successfully Starting the test

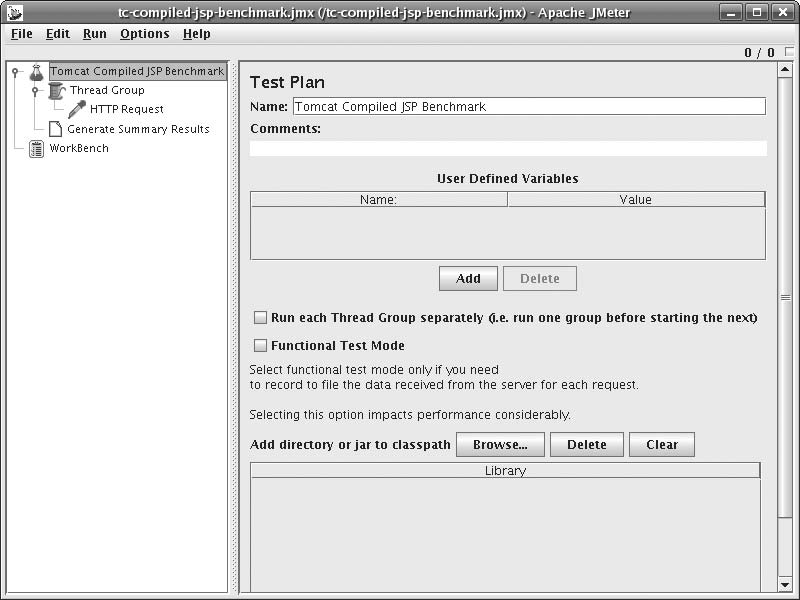
Generate Summary Results = 99979 in 71.0s = 1408.8/s Avg: 38 Min: 0 Max: 25445 Err: 0 (0.00%)

Tidying up ...

... end of run

Notice that the requests per second reported by JMeter (an average of 1408.8 requests per second) is significantly lower than that reported by both ab and siege, for the same hardware, the same version of Tomcat, and the same benchmark. This demonstrates that JMeter’s HTTP client is slower than that of ab and siege. You can use JMeter to find out if a change to your webapp, your Tomcat installation, or your JVM, accelerates or slows the response times of web pages; however, you

cannot use JMeter to determine the server’s maximum number of requests per second that it can successfully serve because JMeter’s HTTP client appears to be slower than Tomcat’s server code.



*Figure 10-1.* *Apache JMeter GUI showing the fully assembled test plan*

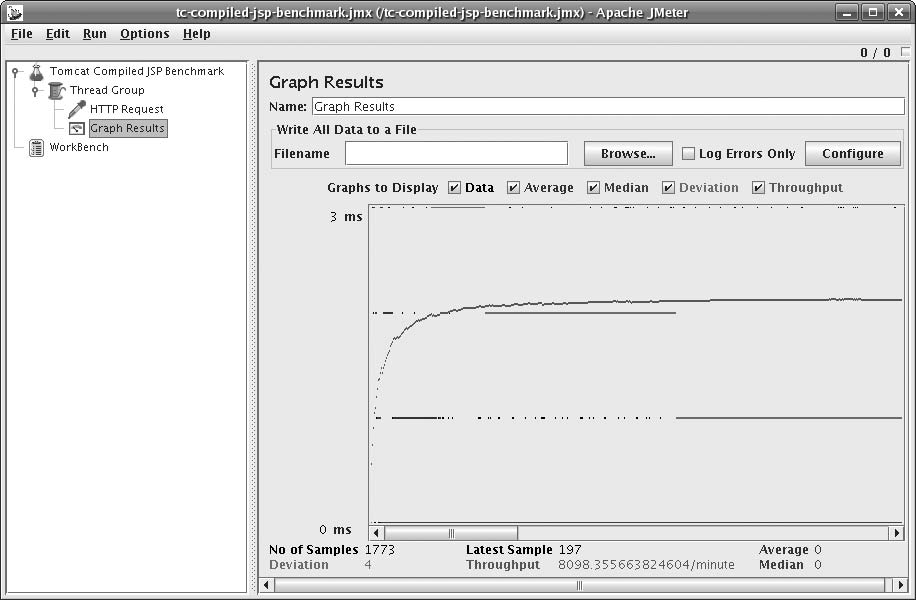
You may also graph the test results in JMeter. To do this, run JMeter in graphical mode again, then:

1. Open the test plan you created earlier.
2. In the tree view, select the Generate Summary Results tree node and delete it (one easy way to do this is to hit the delete key on your keyboard once).
3. Select the Thread Group tree node, then right click on it and select Add ➝ Listener
4. Graph Results.
5. Save your test plan under a new name; this time for graphical viewing of test results.
6. Select the Graph Results tree node.

Now, you’re ready to rerun your test and watch as JMeter graphs the results in real time.

Again, make sure that you give the JMeter JVM enough heap memory so that it does not run its own garbage collector often during the test. Also, keep in mind that the Java VM must spend time graphing while the test is running, which will decrease the accuracy of the test results. How much the accuracy will decrease depends on how fast the computer you’re running JMeter on is (the faster the better). But, if you’re just graphing to watch results in real time as a test is being run, this is a great way to observe.

When you’re ready to run the test, you can either select Run ➝ Start from the top pull-down menu, or you can hit Ctrl-R. The benchmark test will start again, but you will see the results graph being drawn as the responses are collected by JMeter. [Figure 10-2](#_bookmark705) shows the JMeter GUI graphing the test results.



*Figure 10-2. Apache JMeter graphing test results*

You can either let the test run to completion or you can stop the test by hitting Ctrl-. (hold down the Control key and hit the period key). If you stop the test early, it will likely take JMeter some seconds to stop and reap all of the threads in the request Thread Group. To erase the graph before restarting the test, hit Ctrl-E. You can also erase the graph in the middle of a running test, and the test will continue on, plot- ting the graph from that sample onward.

Using JMeter to graph the results gives you a window into the running test so you can watch it and fix any problems with the test and tailor it to your needs before running it on the commandline. Once you think you have the test set up just right, save a test plan that does not Graph Results, but has a Generate Summary Results tree node so that you can run it on the commandline, andthen save

the test plan again under a new name that conveys the kind of test it is and that it is configured to be run from the commandline. Use the results you obtain on the commandline as the authoritative results. Again, the ab benchmark tool gives you more accurate benchmark results but does not offer as many features as JMeter.

JMeter also has many more features that may help you test your webapps in numerous ways. See the online documentation for more information about this great test tool at <http://jakarta.apache.org/jmeter>.

Web Server Performance Comparison

In the previous sections, you read about some HTTP benchmark clients. Now, we show a useful example in Tomcat that demonstrates a benchmark procedure from start to finish and also yields some information that can help you configure Tomcat so that it performs better for your web application.

We benchmarked all of Tomcat’s web server implementations, plus Apache httpd standalone, plus Apache httpd’s modules that connect to Tomcat to see how fast each configuration is at serving static content. For example, is Apache httpd faster than Tomcat standalone? Which Tomcat standalone web server connector implementation is the fastest? Which AJP server connector implementation is the fastest? How much slower or faster is each? We set out to answer these questions by benchmarking different con- figurations, at least for one hardware, OS, and Java combination.

Because benchmark results are highly dependent on the hardware they were run on, and on the versions of all software used at the time, the results can and do change with time. This is because new hardware is different, and new versions of each software package are different, and the performance characteristics of a different combination of hard- ware and/or software change. Also, the configuration settings used in the benchmark affect the results significantly. By the time you read this, the results below will likely be out-of-date. Also, even if you read this shortly after it is published, your hardware and software combination is not likely to be exactly the same as ours. The only way you can really know how your installation of Tomcat and/or Apache httpd will perform on your machine is to benchmark it yourself following a similar benchmark test procedure.

**Tomcat connectors and Apache httpd connector modules**

Tomcat offers implementations of three different server designs for serving HTTP and implementations of the same three designs for serving AJP:

**JIO (****java.io)**

This is Tomcat’s default connector implementation, unless the APR Connector’s libtcnative library is foundat Tomcat startup time. It is also known as “Coyote.” It is a pure Java TCP sockets server implementation that uses the java.io core Java network classes. It is a fully blocking implementation of both HTTP and AJP. Being written in pure Java, it is binary portable to all operating systems that fully support Java. Many people believe this implementation to be slower than Apache httpd mainly because it is written in Java. The assumption there is that Java is always slower than compiled C.

**APR (Apache Portable Runtime)**

This is Tomcat’s default connector implementation if you install Tomcat on Windows via the NSIS installer, but it is not the default connector implementation for most other stock installations of Tomcat. It is implemented as some Java classes that include a JNI wrapper around a small library named libtcnative written in the C programming language, which in turn depends on the Apache Portable Runtime (APR) library. The Apache httpd web server is also implemented in C and uses APR for its network communications. Some goals of this alternate implementation include offering a server implementation that uses the same open source C code as Apache httpd to outperform the JIO connector and also to offer performance that is at least on par with Apache httpd. One drawback is that because it is mainly implemented in C, a single binary release of this Connector cannot run on all platforms such as the JIO connector can. This means that Tomcat administrators need to build it, so a development environment is necessary, and there could be build problems. But, the authors of this Connector justify the extra set up effort by claiming that Tomcat’s web performance is fastest with this Connector implementation. We’ll see for ourselves by benchmarking it.

**NIO (java.nio)**

This is an alternate Connector implementation written in pure Java that uses the java.nio core Java network classes that offer non blocking TCP socket features. The main goal of this Connector design is to offer Tomcat administrators a Connector implementation that performs better than the JIO Connector by using fewer threads by implementing parts of the Connector in a non blocking fashion. The fact that the JIO Connector blocks on reads and writes means that if the administrator configures it to handle 400 concurrent connections, the JIO Connector must spawn 400 Java threads. The NIO Connector, on the other hand, needs only one thread to parse the requests on many connections, but then each request that gets routed to a servlet must run in its own thread (a limitation mandated by the Java Servlet Specification). Since part of the request handling is done in nonblocking Java code, the time it takes to handle that part of the request is time that a Java thread does not need to be in use, which means a smaller thread pool can be used to handle the same number of concurrent requests. A smaller thread pool usually means lower CPU utilization, which in turn usually means better performance. The theory behind why this would be faster builds on a tall stack of assumptions that may or may not apply to anyone’s own webapp and traffic load. For some, the NIO Connector could perform better, and for others, it could perform worse, as is the case for the other Connector designs.

Alongside these Tomcat Connectors, we benchmarked Apache httpd in both prefork and worker Multi-Process Model (MPM) build configurations, plus configurations of httpd prefork and worker where the benchmarked requests were being sent from Apache httpd to Tomcat via an Apache httpd connector module. We benchmarked the following Apache httpd connector modules:

**mod\_jk**

This module is developed under the umbrella of the Apache Tomcat project. It began years before Apache httpd’s mod\_proxy included support for the AJP pro- tocol (Tomcat’s AJP Connectors implement the server side of the protocol). This is an Apache httpd module that implements the client end of the AJP protocol. The AJP protocol is a TCP packet-based binary protocol with the goal of relaying the essentials of HTTP requests to another server software instance significantly faster than could be done with HTTP itself. The premise is that HTTP is very plain-text oriented, and thus requires slower, more complex parsers on the server side of the connection, and that if we instead implement a binary protocol that relays the already-parsed text strings of the requests, the server can respond significantly faster, and the network communications overhead can be

minimized. At least, that’s the theory. We’ll see how significant the difference is. As of the time of this writing, most Apache httpd users who add Tomcat to their web servers to support servlets and/or JSP, build and use mod\_jk mainly because either they believe that it is significantly faster than mod\_proxy, or because they do not realize that mod\_proxy is an easier alternative, or because someone suggested mod\_jk to them. We set out to determine whether building, installing, configuring, and maintaining mod\_jk was worth the resulting performance.

**mod\_proxy\_ajp**

This is mod\_proxy’s AJP protocol connector support module. It connects with Tomcat via TCP to Tomcat’s AJP server port, sends requests through to Tom- cat, waits for Tomcat’s responses, andthen Apache httpd forwards the responses to the web client(s). The requests go through Apache httpd to Tomcat and back, and the protocol used between Apache httpd and Tomcat is the AJP protocol, just as it is with mod\_jk. This connector became part of Apache httpd itself as of httpd version 2.2 andis already built into the httpd that comes with most operating systems (or it is prebuilt as a loadable httpd module). No extra compilation or installation is usually necessary to use it —just configuration of Apache httpd. Also, this module is a derivative of mod\_jk, so mod\_proxy\_ajp’s code and features are very similar to those of mod\_jk.

**mod\_proxy\_http**

This is mod\_proxy’s HTTP protocol connector support module. Like *mod\_ proxy\_ajp*, it connects with Tomcat via TCP, but this time it connects to Tomcat’s HTTP (web) server port. A simple way to think about how it works: the web client makes a request to Apache httpd’s web server, and then httpd makes that same request on Tomcat’s web server, Tomcat responds, and httpd forwards the response to the web client. All communication between Apache httpd and Tomcat is done via HTTP when using this module. This connector module is also part of Apache httpd, and it usually comes built into the httpd binaries found on most operating systems. It has been part of Apache httpd for a very long time, so it is available to you regardless of which version of Apache httpd you run.

**Benchmarked hardware and software configurations**

Because one of the machines is a desktop machine and the other is a laptop, the results of this benchmark also show the difference in static file serving capability between a single processor laptop and a dual processor desktop. We are not attempting to match up the two different CPU models in terms of processing power similarity, but instead we benchmarked a typical dual CPU desktop machine versus a typical single processor laptop, both new (retail-wise) aroundthe time of the benchmark. Also, both machines have simple ext3 hard disk partitions on the hard disks, so no LVM or RAID configurations were used on either machine for these benchmarks.

Both of these machines are x86\_64 architecture machines, but their CPUs were designed and manufactured by different companies. Also, both of these machines came equipped with gigabit Ethernet, and we benchmarked them from another fast machine that was also equipped with gigabit Ethernet, over a network switch that supported gigabit Ethernet.

We chose to use the ApacheBench (ab) benchmark client. We wanted to make sure that the client supported HTTP 1.1 keep-alive connections because that’s what we wante dto benchmark and that the client was fast enough to give us the most accurate results. Yes, we are aware of Scott Oaks’s blog article about ab (read it at [http://](http://weblogs.java.net/blog/sdo/archive/2007/03/ab_considered_h.html) [weblogs.java.net/blog/sdo/archive/2007/03/ab\_considered\_h.html](http://weblogs.java.net/blog/sdo/archive/2007/03/ab_considered_h.html)). While we agree with

Mr. Oaks on his analysis of how ab works, we carefully monitored the benchmark client’s CPU

utilization and ensured that ab never saturated the CPU it was using during the benchmarks we ran. We also turned up ab’s concurrency so that more than one HTTP request could be active at a time. The fact that a single ab process can use exactly one CPU is okay because the operating system performs context switching on the CPU faster than the network can send and receive request and response packets. Per CPU, everything is actually a single stream of CPU instructions on the hardware anyway, as it turns out. With the hardware we used for our benchmarks, the web server machine did not have enough CPU cores to saturate ab’s CPU, so we really did benchmark the performance of the web server itself.

We’re testing Tomcat version 6.0.1 (this was the latest release available when we began benchmarking—we expect newer versions to be faster, but you never know until you benchmark it) running on Sun Java 1.6.0 GA release for x86\_64, Apache version 2.2.3, mod\_jk from Tomcat Connectors version 1.2.20, andthe APR connec- tor (libtcnative) version 1.1.6. At the time of the benchmark, these were the newest versions available—sorry we cannot benchmark newer versions for this book, but the great thing about well-detailed benchmarks is that they give you enough information to reproduce the test yourself. The operating system on both machines was Fedora Core 6 Linux x86\_64 with updates applied via yum. The kernel version was 2.6.18.2.

Tomcat’s JVM startup switch settings were:

*-Xms384M -Xmx384M -Djava.awt.headless=true -Djava.net.preferIPv4Stack=true*

Here is our Tomcat configuration for the tests: Stock conf/web.xml. Stock conf/server. xml, except that the access logger was not enabled (no logging per request), and these connector configs, which were enabled one at a time for the different tests:

*<!-- The stock HTTP JIO connector. -->*

*<Connector port="8080" protocol="HTTP/1.1" maxThreads="150" connectionTimeout="20000" redirectPort="8443" />*

*<!-- The HTTP APR connector. -->*

*<Connector port="8080"*

*protocol="org.apache.coyote.http11.Http11AprProtocol" enableLookups="false" redirectPort="8443" connectionTimeout="20000"/>*

*<!-- HTTP NIO connector. -->*

*<Connector port="8080"*

*maxThreads="150" connectionTimeout="20000" redirectPort="8443" protocol="org.apache.coyote.http11.Http11NioProtocol"/>*

*<!-- AJP JIO/APR connector, switched by setting LD\_LIBRARY\_PATH. -->*

*<Connector port="8009" protocol="AJP/1.3" redirectPort="8443" />*

*<!-- AJP NIO* *connector. -->*

*<Connector protocol="AJP/1.3" port="0" channelNioSocket.port="8009"* *channelNioSocket.maxThreads="150" channelNioSocket.maxSpareThreads="50" channelNioSocket.minSpareThreads="25" channelNioSocket.bufferSize="16384"/>*

The APR code was enabled by using the HTTP APR connector configuration shown, plus setting and exporting LD\_LIBRARY\_PATH to a directory containing libtcnative in the Tomcat JVM process’s environment, and then restarting Tomcat.

We built the APR connector like this:

*# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse -mmmx -msse -msse2 - msse3 -m3dnow -mtune=athlon64" ./configure --with-apr=/usr/bin/apr-1-config -- prefix=/opt/tomcat/apr-connector*

*# make && make install*

We used the same CFLAGS when building Apache httpd and mod\_jk. Here’s how we built and installed mod\_jk:

*# cd tomcat-connectors-1.2.20-src/native*

*# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse -mmmx -msse -msse2 - msse3 -m3dnow -mtune=athlon64" ./configure --with-apxs=/opt/httpd/bin/apxs*

*[lots of configuration output removed]*

*# make && make install*

This assumes that the root directory of the Apache httpd we built is /opt/httpd. We built the APR connector, httpd, and mod\_jk with GCC 4.1.1:

# gcc --version

gcc (GCC) 4.1.1 20061011 (Red Hat 4.1.1-30)

Copyright (C) 2006 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

We downloaded Apache httpd version 2.2.3 from [http://httpd.apache.org](http://httpd.apache.org/) and built it two different ways and benchmarked each of the resulting binaries. We built it for pre- fork MPM and worker MPM. These are different multithreading and multiprocess models that the server can use. Here are the settings we used for prefork and worker MPM:

*# prefork MPM*

*<IfModule prefork.c> StartServers 8*

*MinSpareServers 5*

*MaxSpareServers 20*

*ServerLimit 256*

*MaxClients 256*

*MaxRequestsPerChild 4000*

*</IfModule>*

*#* *worker MPM*

*<IfModule worker.c> StartServers 3*

*MaxClients 192*

*MinSpareThreads 1*

*MaxSpareThreads 64*

*ThreadsPerChild 64*

*MaxRequestsPerChild 0*

*</IfModule>*

We disabled Apache httpd’s common access log so that it would not needto log anything per each request (just as we configured Tomcat). And, we turned on Apache httpd’s KeepAlive configuration option:

*KeepAlive On MaxKeepAliveRequests 100*

*KeepAliveTimeout 5*

We enabled *mod\_proxy* one of two ways at a time. First, for proxying via HTTP:

*ProxyPass /tc http://127.0.0.1:8080/ ProxyPassReverse /tc http://127.0.0.1:8080/*

Or, for proxying via AJP:

*ProxyPass /tc ajp://127.0.0.1:8009/ ProxyPassReverse /tc ajp://127.0.0.1:8009/*

And, we configured mod\_jk by adding this to httpd.conf:

*LoadModule jk\_module /opt/httpd/modules/mod\_jk.so JkWorkersFile /opt/httpd/conf/workers.properties JkLogFile /opt/httpd/logs/mod\_jk.log*

*JkLogLevel info*

*JkLogStampFormat "[%a %b %d %H:%M:%S %Y] "*

*JkOptions +ForwardKeySize +ForwardURICompat* *-ForwardDirectories JkRequestLogFormat "%w %V %T"*

*JkMount /tc/\* worker1*

Plus we created a *workers.properties* file for *mod\_jk* at the path we specified in the

httpd.conf file:

*worker.list=worker1 worker.worker1.type=ajp13 worker.worker1.host=localhost*

*worker.worker1.port=8009 worker.worker1.connection\_pool\_size=150 worker.worker1.connection\_pool\_timeout=600 worker.worker1.socket\_keepalive=1*

Of course, we enabled only one Apache httpd connector module at a time in the configuration.

**Benchmark procedure**

We benchmarked two different types of static resource requests: small text files and 9k image files. For both of these types of benchmark tests, we set the server to be able to handle at least 150 concurrent client connections, and set the benchmark client to open no more than 149 concurrent connections so that it never attempted to use more concurrency than the server was configured to handle. We set the bench- mark client to use HTTP keep-alive connections for all tests.

For the small text files benchmark, we’re testing the server’s ability to read the HTTP request and write the HTTP response where the response body is very small. This mainly tests the server’s ability to respond fast while handling many requests concurrently. We set the benchmark client to request the file 100,000 times, with a possible maximum of 149 concurrent connections. This is how we created the text file:

*$ echo 'Hello world.' > test.html*

We copied this file into Tomcat’s ROOT webapp and also into Apache httpd’s document root directory.

Here is the ab command line showing the arguments we used for the small text file benchmark tests:

*$ ab -k -n 100000 -c 149* [*http://192.168.1.2/test.html*](http://192.168.1.2/test.html)

We changed the requested URL appropriately for each test so that it made requests that would benchmark the server we intended to test each time.

For the 9k image files benchmark, we’re testing the server’s ability to serve a larger amount of data in the response body to many clients concurrently. We set the benchmark client to request the file 20,000 times, with a possible maximum of 149 concurrent connections. We specified a lower total number of requests for this test because the size of the data was larger, so we adjusted the number of requests down to compensate somewhat, but still left it high to place a significant load on the server. This is how we created the image file:

*$ dd if=a-larger-image.jpg of=9k.jpg bs=1 count=9126*

We chose a size of 9k because if we went much higher, both Tomcat and Apache httpd would easily saturate our 1 Mb Ethernet link between the client machine and the server machine. Again, we copied this file into Tomcat’s ROOT webapp and also into Apache httpd’s document root directory.

Here is the ab command line showing the arguments we used for the small text file benchmark tests:

*$ ab -k -n 20000 -c 149* [*http://192.168.1.2/20k.jpg*](http://192.168.1.2/20k.jpg)

For each invocation of ab, we obtained the benchmark results by following this procedure:

1. Configure and restart the Apache httpd and/or Tomcat instances that are being tested.
2. Make sure the server(s) do not log any startup errors. If they do, fix the problem before proceeding.
3. Run one invocation of the ab command line to get the servers serving their first requests after the restart.
4. Run the ab command line again as part of the benchmark.
5. Make sure that ab reports that there were zero errors and zero non-2xx responses, when all requests are complete.
6. Wait a few seconds between invocations of ab so that the servers go back to an idle state.
7. Note the requests per second in the ab statistics.
8. Go back to step 4 if the requests per second change significantly; otherwise, this iteration’s requests per second are the result of the benchmark. If the numbers continue to change

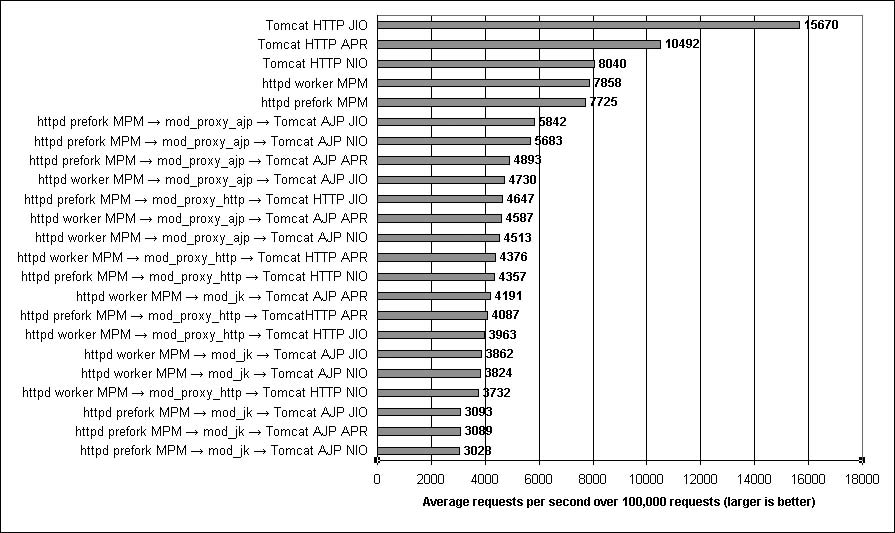
significantly, give up after 10 iterations of ab, and record the last requests per second value as the benchmark result.

The idea here is that the servers will be inefficient for the first couple or few invocations of ab, but then the server software arrives at a state where everything is well initialized. The Tomcat JVM begins to profile itself and natively compile the most heavily used code for that particular use of the program, which further speeds response time. It takes a few ab invocations for the servers to settle into their more optimal runtime state, and it is this state that we should be benchmarking—the state the servers would be in if they were serving for many hours or days as production servers tend to do.

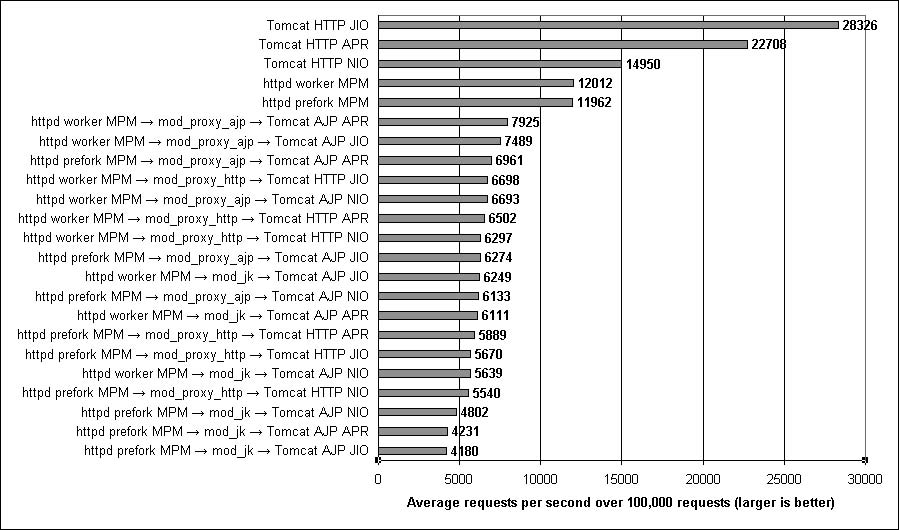
**Benchmark results and summary**

We ran the benchmarks and graphed the results data as bar charts, listing the web server configurations in descending performance order (one graph per test per computer). First, we look at how the machines did in the small text files benchmark (see Figures [10-3](#_bookmark747) and [10-4).](#_bookmark748)

Notice that Figures [10-3](#_bookmark747) [and 10-4](#_bookmark748) look very similar. On both machines, Tomcat standalone JIO is the fastest web server for serving these static text files, followed by APR, followed by NIO. The two build configurations of Apache httpd came in fourth and fifth fastest, followed by all of the permutations of Apache httpd connected to Tomcat via a connector module. And, dominating the slow end of the graphs is mod\_jk.



*Figure 4-3. Benchmark results for serving small text files on the AMD64 laptop*

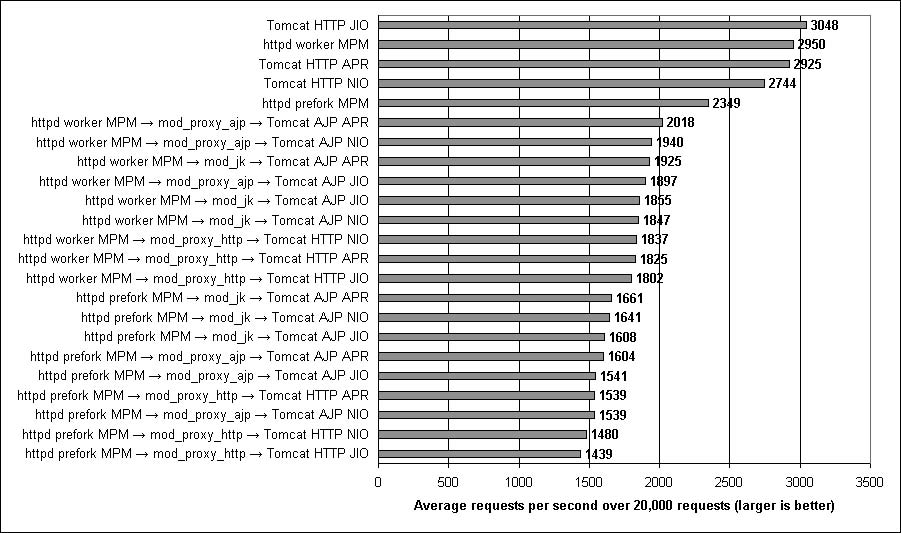


*Figure 10-4. Benchmark results for serving small text files on the EM64T tower*

It is also interesting to compare the requests per second results for one web server configuration on both graphs. The AMD64 laptop has one single core processor, and the EM64T has two single core processors; thus, if dual EM64T computer works efficiently, and if the operating system and JVM can effectively take advantage of both processors, the dual EM64T computer should be able to sustain slightly less than double the requests per second that the single processor AMD64 machine could. Of course, this assumes that the two processor models are equally fast at executing

instructions; they may not be. But, comparing the results for the two computers, the same web server configuration on the dual EM64T computer does sustain nearly double the requests per second, minus a percent for the overhead of the two processors sharing one set of system resources, such as RAM, data and I/O buses, and so on. This one computer with two processors in it can handle nearly the same number of requests that two single processor computers can, and both Tomcat and Apache httpd are able to take advantage of that.

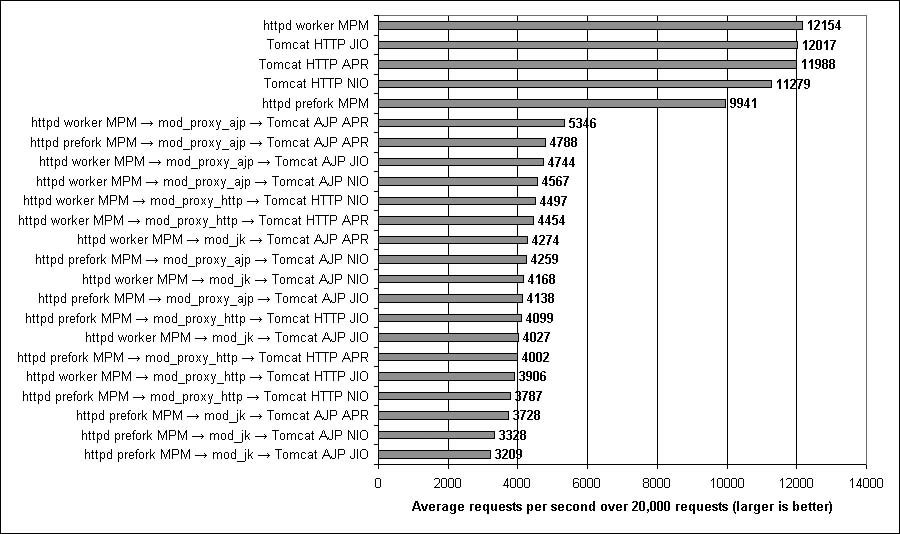
Next, we examine the results of the 9k image files benchmark on both machines. Fig- ures [10-5](#_bookmark749) [and 10-6](#_bookmark754) show the results for the AMD64 computer and the dual EM64T computer, respectively.



*Figure 10-5. Benchmark* *results for serving 9k* *image files on* *the AMD64* *laptop*

In [Figure 10-5,](#_bookmark749) you can see that on AMD64, Tomcat standalone JIO wins again, with Apache httpd worker MPM trailing close behind. In this benchmark, their performance is nearly identical, with Tomcat standalone APR in a very close third place. Tomcat standalone NIO is in fourth place, trailing a little behind APR. Apache httpd prefork MPM is fifth fastest again behind all of the Tomcat standalone configurations. Slower still are all of the permutations of Apache httpd connecting to Tomcat via connector modules. This time, we observed mod\_jk perform about average among the connector modules, with some configurations of mod\_proxy\_http performing the slowest.

[Figure 4-6](#_bookmark754) is somewhat different, showing that on the dual EM64T, Apache httpd edges slightly ahead of Tomcat standalone’s fastest connector: JIO. The difference in performance between the two is very small—about 1 percent. This may hint that there is a difference in how EM64T behaves versus AMD64. It appears that Apache httpd is 1 percent faster than Tomcat on EM64T when serving the image files, at least on the computers we benchmarked. You should not assume this is the case with newer computers, as many hardware details change! Also, we observed all three Tomcat standalone connectors performing better than Apache httpd prefork in this set of benchmarks. The configurations where Apache httpd connects to Tomcat via a connector module were again the slowest performing configurations, with mod\_jk performing the slowest.



*Figure 4-6. Benchmark results for serving 9k image files on the EM64T tower*

Does the dual EM64T again serve roughly double the number of requests per second as the single processor AMD64 when serving the image files? No. For some reason, it’s more like four times the number of requests per second. How could it be possible that by adding one additional processor, the computer can do four times the work? It probably can’t. The only explanation we can think of is that something is slowing down the AMD64 laptop’s ability to serve the image files to the processor’s full potential. This isn’t necessarily a hardware problem; it could be that a device driver in this version of the kernel is performing inefficiently and slowing down the benchmark. This hints that the benchmark results for the 9k image benchmark on the AMD64 computer may not be accurate due to a slow driver. However, this is the observed performance on that computer. Until and unless a different kernel makes it perform better, this is how it will perform. Knowing that, it is unclear whether Tom- cat or Apache httpd is faster serving the 9k image files, although we would guess that the EM64T benchmark results are more accurate.

Here is a summary of the benchmark results, including some important stats:

* Tomcat standalone was faster than Apache httpd compiled for worker MPM in all of our benchmark tests except the 9k image benchmark test on Intel 64-bit Xeon, and even in that benchmark, httpd was only 1 percent faster than Tom- cat. We observed that Tomcat standalone JIO was almost always the fastest way to serve static resources. Tomcat served them between 3 percent and 136 percent faster than Apache httpd in our benchmarks—Tomcat standalone JIO was a minimum of 3 percent faster than Apache httpd (worker MPM) for 9k image files, except for the Intel 64-bit Xeon benchmark, where httpd appeared to per- form 1 percent faster than Tomcat. But in the small files benchmark, Tomcat was a minimum of 99 percent faster than Apache httpd anda maximum of 136 percent faster than Apache httpd.
* Apache httpd built to use worker MPM was the fastest configuration of Apache httpd we tested; Apache httpd built to use prefork MPM was slower than worker MPM in all of our standalone tests. We observed worker MPM serving a minimum of 0.4 percent faster than prefork MPM and a maximum of 26 percent faster than prefork MPM. There was almost no difference in performance between the two in our small text files benchmarks, but in the 9k image files benchmark, the difference was at least 22 percent.
* Tomcat standalone (configured to use any HTTP connector implementation) was always faster than Apache httpd built and configured for prefork MPM; Tomcat standalone was a minimum of 21 percent faster than Apache httpd anda maximum of 30 percent faster than Apache httpd for 9k image files, and for small files Tomcat was a minimum of 103 percent

faster than Apache httpd anda maximum of 136 percent faster than Apache httpd prefork MPM.

* Apache httpd was quite a bit slower at serving small files. Tomcat standalone’s JIO, APR, and NIO connectors were each faster than Apache httpd—Tomcat’s JIO connector performedas much as 136 percent faster than Apache httpd’s fastest configuration, Tomcat’s APR connector performed 89 percent faster than Apache httpd, and Tomcat 6.0’s NIO connector performed 25 percent faster than Apache httpd. In this common use case benchmark, Apache httpd dropped to fourth place behind all of Tomcat standalone’s three HTTP connectors.
* Serving Tomcat’s resources through Apache httpd was very slow compared to serving them directly from Tomcat. When we compared the benchmark results between Tomcat standalone and Tomcat serving through Apache httpd via mod\_ proxy, Tomcat standalone consistently served at least 51 percent faster when using only Tomcat’s JIO connector without Apache httpd. (including all three Apache httpd connector modules: mod\_jk, mod\_proxy\_ajp, and mod\_proxy\_ http). In the small text files benchmark, Tomcat standalone was a minimum of 168 percent faster than the Apache httpd to Tomcat configurations anda maxi- mum of 578 percent faster! That’s not a misprint—it’s really 578 percent faster. For the 9k image files benchmark, Tomcat standalone was at least 51 percent faster and at most 274 percent faster.
* AJP outperformed HTTP when using mod\_proxy. The benchmark results show that mod\_proxy\_ajp was consistently faster than mod\_proxy\_http. The margin between the two protocols was as low as 1 percent and as high as 30 percent when using the same Tomcat connector design, but it was usually smaller, with mod\_proxy\_ajp averaging about 13 percent faster than mod\_proxy\_http.
* Serving Tomcat’s static resources through an Apache httpd connector module was never faster than serving the same static resources through just Apache httpd by itself. The benchmark results of serving the resources through an httpd connector module (from Tomcat) were always somewhat slower than just serving the static resources straight from Apache httpd. This means that benchmarking Apache httpd standalone will tell you a number slightly higher than the theoretical maximum that you could get by serving the same resource(s) through an httpd connector module. This also means that no matter how performant Tomcat is, serving its files through Apache httpd throttles Tomcat down so that Tom- cat is slower than Apache httpd.
* mod\_jk was not faster than mod\_proxy, except in the 9k image benchmark and then only on AMD64. In our tests, serving Tomcat’s resources through Apache httpd via mod\_jk was only faster than using mod\_proxy on the AMD64 laptop and only in the 9k image benchmark. In all the other benchmarks, mod\_jk was slower than mod\_proxy\_ajp.

How is it possible for pure-Java Tomcat to serve static resource faster than Apache httpd? The main reason we can think of: because Tomcat is written in Java and because Java bytecode can be natively compiled and highly optimized at runtime, well-written Java code can run very fast when it runs on a mature Java VM that implements many runtime optimizations, such as the Sun Hotspot JVM. After it runs and serves many requests, the JVM knows how to optimize it for that particular use on that particular hardware. On the other hand, Apache httpd is written in C, which is completely compiled ahead of runtime. Even though you can tell the compiler to heavily optimize the binaries, no runtime optimizations can take place. So, there is no opportunity with Apache httpd to take advantage of the many runtime optimizations that Tomcat enjoys.

Another potential reason Tomcat serves the web faster than Apache httpd is that every release of Sun’s JVM seems to run Java code faster, which has gone on for many release cycles of their JVM. That means that even if you’re not actively changing your Java program to perform better, it will likely keep improving every time you run it on a newer, faster JVM if the same progress on JVM performance continues. This does, however, make the assumption that newer JVMs will be compatible enough to run your Java program’s bytecode without any modifications.

**What else we could have benchmarked**

In this benchmark, we tested the web server’s performance when serving HTTP. We did not benchmark HTTPS (encrypted HTTP). The performance characteristics are probably significantly different between HTTP and HTTPS because with HTTPS, both the server and the client must encrypt and decrypt the data in both directions over the network. The overhead caused by the encryption slows down the requests and responses to varying degrees on different implementations of the crypto code. We have not benchmarked the HTTPS performance of the above web server configurations. Without benchmarking it, many believe that Apache httpd’s HTTPS performance is higher than that of Tomcat, and usually people base that belief on the idea that C code is faster than Java code. Our HTTP benchmark disproves that in three out of our four benchmark scenarios, and the fourth one is not significantly better on the C side. We do not know which web server configuration would be fastest serving HTTPS without benchmarking them. But, if either the C encryption code or the Java encryption code is the fastest—by a significant margin—Tomcat implements both because you can configure the APR connector to use OpenSSL for HTTPS encryption, which is the same C library that Apache httpd uses.

We could have benchmarked other metrics such as throughput; there are many more interesting things to learn by watching any particular metric that ab reports. For this benchmark, we define greater performance to mean a higher number of requests per second being handled successfully (a 2xx response code).

We could have benchmarked other static file sizes, including files larger than 9k in size, but with files as large as 100k, all of the involved server configurations saturate the bandwidth of a megabit Ethernet network. This makes it impossible to measure how fast the server software itself could serve the files because the network was not fast enough. For our test, we did not have network bandwidth greater than 1 Mb Ethernet.

We could have tested with mixed file sizes per HTTP request, but what mixture would we choose, and what use case would that particular mixture represent? The results of benchmarks such as these would only be interesting if your own web traffic had a similar enough mixture, which is unlikely. Instead, we focused on bench- marking two file sizes, one file size per benchmark test.

We could have tested with a different number of client threads, but 150 threads is the default (as of this writing) on both Tomcat and Apache httpd, which means many administrators will use these settings—mainly due to lack of time to learn what the settings do and how to change them in a useful way. We ended up raising some of the limits on the Apache httpd side to try to find a way to make httpd perform better when the benchmark client sends a maximum of 149 concurrent requests; it worked.

There are many other things we could have benchmarked and many other ways we could have benchmarked. Even covering other common use cases is beyond the scope of this book. We’re trying to show only one example of a benchmark that yields some useful information about how the performance of Tomcat’s web server implementations compares with that of Apache httpd in a specific limited environment and for specific tests.

**External Tuning**

Once you’ve got an idea how your application and Tomcat instance respond to load, you can begin some performance tuning. There are two basic categories of tuning detailed here:

**External tuning**

Tuning that involves non-Tomcat components, such as the operating system that Tomcat runs on and the Java virtual machine running Tomcat.

**Internal tuning**

Tuning that deals with Tomcat itself, ranging from changing settings in configuration files to modifying the Tomcat source code. Modifications to your web application also fall into this category.

In this section, we detail the most common areas of external tuning, and then move on to internal tuning in the next section.

**JVM Performance**

Tomcat doesn’t run directly on a computer; there is a JVM and an operating system between it and the underlying hardware. There are relatively few complete and fully compatible Java virtual machines to choose from for any given operating system and architecture combination, so most people will probably stick with Sun’s or their own operating system vendor’s implementation.

If your goal is to run the fastest Java runtime and squeeze the most performance out of your webapp, you should benchmark Tomcat and your webapp on each of the Java VMs that are available for your hardware and operating system combination. Do not assume that the Sun Java VM is going to be the fastest because that is often not the case (at least in our experience). You should try other brands and even different major version numbers of each brand to see what runs your particular webapp fastest.

If you choose just one version of the Java class file format that JVMs you use must support (for example, you want to compile your webapp for Java 1.6 JVMs), you can benchmark each available JVM brand that supports that level of the bytecodes, and choose one that best fits your needs. For instance, if you choose Java 1.6, you could benchmark Sun’s 1.6 versus IBM’s 1.6 versus BEA’s 1.6. One of these will run Tomcat and your webapp the fastest. All of these brands are used in production by a large number of users and are targeted at slightly different user bases. As a generic example of performance improvements between major versions of one JVM brand, a major version

upgrade could buy you a 10 percent performance increase. That is, upgrading from a Java 1.5 JVM to a Java 1.6 JVM your webapp may run 10 percent faster, without changing any code in it whatsoever. This is a ballpark figure, not a benchmark result; your mileage may vary, depending on the brands and versions you test and what your webapp does.

It is likely true that newer JVMs have both better performance and less stability, but the longer a major version of the JVM has been released as a final/stable version, the less you have to worry about its stability. A good rule of thumb is to get the latest stable version of the software, except when the latest stable version is the first or secondstable release of the next major version of the software. For example, if the latest stable version is 1.7.0, you may opt for 1.6.29 instead if it is more stable and performs well enough.

It is often the case that people try to modify the JVM startup switches to make their Tomcat JVM serve their webapp’s pages faster. This can help, but does not usually yield a high percentage increase in performance. The main reason it does not help much: the JVM vendor did their own testing before releasing the JDK, found which settings yield the best performance, and made those settings the defaults.

If you change a JVM startup switch to activate a setting that is not the default, chances are that you will slow down your JVM. You have been warned! But, in case you would like to see which Sun JVM settings you could change, have a look at [http://www.md.pp.ru/~eu/](http://www.md.pp.ru/%7Eeu/jdk6options.html) [jdk6options.html](http://www.md.pp.ru/%7Eeu/jdk6options.html).

One exception here is the JVM’s heap memory allocation. By default, vendors choose for the JVM to start by allocating a small amount of memory (32 MB in the Sun JVM’s case), and if the Java application requires more memory, the JVM’s heap size is reallocated larger while the application is paused. The JVM may do this a number of times in small memory increments before it hits a heap memory size ceiling. Because the application is paused each time the heap size is increased, performance suffers. If that is happening while Tomcat is serving a webapp’s pages, the page responses will appear to take far longer than normal to all web clients whose requests are outstanding at the time the pause begins. To avoid these pauses, you can set the minimum heap size and the maximum heap size to be the same. That way, the JVM will not attempt to expand the heap size during runtime. To do this to Tomcat’s JVM startup switches, just set the JAVA\_OPTS environment variable to some- thing such as -*Xms512M -Xmx512M*. (This means that the maximum and minimum heap size should be set to 512 MB.) Set the size to an appropriate value on your machine, based on how much memory it has free after it boots.

You can also try benchmarking different garbage collection algorithm settings, however, as we stated earlier you may find that the default settings are always fastest. You never know until you benchmark it, though. Check the documentation for the JVM you’re benchmarking to find the startup switch that will enable a different garbage collection algorithm because these settings are JVM implementation-specific. Again, you’ll want to set it in JAVA\_OPTS to get Tomcat to start the JVM that way.

**Operating System Performance**

Andwhat about the OS? Is your server operating system optimal for running a large, high-volume web server? Of course, different operating systems have very different design goals. OpenBSD, for example, is aimed at security, so many of the limits in the kernel are set small to prevent various forms of denial-of-service attacks (one of OpenBSD’s mottoes is “Secure by default”). These limits will most likely need to be increased to run a busy web server.

Linux, on the other hand, aims to be easy to use, so it comes with the limits set higher. The BSD kernels come out of the box with a “generic” kernel, that is, most of the drivers are statically linked in. This makes it easier to get started, but if you’re building a custom kernel to raise some of those

limits, you might as well rip out unneeded devices. Linux kernels have most of the drivers dynamically loaded. On the other hand, memory itself is getting cheaper, so the reasoning that led to loadable device drivers is less important. What is important is to have lots and lots of memory and to make a lot of it available to the server.

Memory is cheap these days, but don’t buy cheap memory—brand name memory costs only a little more and repays the cost in reliability.

If you run any variant of Microsoft Windows, be sure you have the server version (e.g., Windows Vista Server instead of just Windows Vista Pro). In other non server versions, the end user license agreement and/or the operating system’s code itself may restrict the number of users, or the number of network connections that you can use, or place other restrictions on what you can run. Additionally, be sure you obtain the latest Microsoft service packs frequently, for the obvious security reasons (this is true for any system, but is particularly important for Windows).

**Internal Tuning**

This section details a specific set of techniques that will help your Tomcat instance run faster, regardless of the operating system or JVM you are using. In many cases, you may not have control

of the OS or JVM on the machine you are deploying to. In those situations, you should still make recommendations in line with what was detailed in the last section; however, you still should be able to affect changes in Tomcat itself. Here is where we think are the best places to start internally tuning Tomcat.

**Disabling DNS Lookups**

When a web application wants to log information about the client, it can either log the client’s numeric IP address or look up the actual host name in the Domain Name Service data. DNS lookups require network traffic, involving a round-trip response from multiple servers, possibly far away and possibly inoperative, resulting in delays. To disable these delays you can turn off DNS lookups. Then, whenever a web appli- cation calls the *getRemoteHost*( ) method in the HTTP request object, it will only get the numeric IP address. This is set in the Connector object for your application, in Tomcat’s server.xml file. For the common java.io HTTP 1.1 connector, use the enableLookups attribute. Just find this part of the server.xml file:

*<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->*

*<Connector port="8080" maxHttpHeaderSize="8192"*

*maxThreads="150" minSpareThreads="25" maxSpareThreads="75" enableLookups="true" redirectPort="8443" acceptCount="100" connectionTimeout="20000" disableUploadTimeout="true" />*

Just change the enableLookups value from "true" to "false", andrestart Tomcat. No more DNS lookups and their resulting delays!

Unless you need the fully qualified hostname of every HTTP client that connects to your site, we recommend turning off DNS lookups on production sites. Remember that you can always look up the names later, outside of Tomcat. Not only does turning them off save network bandwidth, lookup time, and memory, but in sites where quite a bit of traffic generates quite a bit of log data, it may save a noticeable amount of disk space as well. For low traffic sites, turning off DNS lookups may not have as dramatic an effect, but it is still not a bad practice. How often have low traffic sites become high traffic sites overnight?

**Adjusting the Number of Threads**

Another performance control on your application’s Connector is the number of request handler

threads it uses. By default, Tomcat uses a thread pool to provide rapid response to incoming requests. A thread in Java (as in other programming languages) is a separate flow of control, with its own interactions with the operating system, and its own local memory—but with some memory shared among all threads in the process. This allows developers to provide fine-grained organization of code that will respond well to many incoming requests.

You can control the number of threads that are allocated by changing a Connector’s *minThreads* and *maxThreads* values. The values provided are adequate for typical installations but may need to be increased as your site gets larger. The *minThreads* value should be high enough to handle a minimal loading. That is, if at a slow time of day you get five hits per second and each request takes under a second to process, the five preallocated threads are all you will need. Later in the day, as your site gets busier, more threads will need to be allocated (up to the number of threads specified in *maxThreads* attribute). There needs to be an upper limit to prevent spikes in traffic (or a denial-of-service attack from a malicious user) from bombing out your server by making it exceed the maximum memory limit of the JVM.

The best way to set these to optimal values is to try many different settings for each and test them with simulated traffic loads while watching response times and memory utilization. Every machine, operating system, and JVM combination may act differently, and not everyone’s web site traffic volume is the same, so there is no cut- and-dry rule on how to determine minimum and maximum threads.

**S****peeding Up JSPs**

When a JSP is first accessed, it is converted into Java servlet source code, which must then be compiled into Java bytecode.

Another option is to not use JSPs altogether and take advantage of some of the various Java templating engines available today. While this is obviously a larger scale decision, many have found it worth at least investigating. For detailed information about other templating languages that you can use with Tomcat.

**Precompiling JSPs by requesting them**

Since a JSP is normally compiled the first time it’s accessed via the web, you may wish to perform precompilation after installing an updated JSP instead of waiting for the first user to visit it. Doing so helps to ensure that the new JSP works as well on your production server as it did on your test machine.

There is a script file called jspc in the Tomcat bin/ directory that looks as though it might be used to precompile JSPs, but it is not. It does run the translation phase from JSP source to Java source, but not the Java compilation phase, and it generates the resulting Java source file in the current directory, not in the work directory for the web application. It is primarily for the benefit of people debugging JSPs.

The simplest way to ensure precompilation of any given JSP file is to simply access the JSP through a web client. This will ensure the file is translated to a servlet, compiled, and then run. It also has the advantage of exactly simulating how a user would access the JSP, allowing you to see what they would. You can catch any errors, correct them, and then repeat the process. Of course, this development cycle is best done in a development environment, not on the production server.

**Precompiling JSPs at webapp start time**

Another excellent but seldomly used feature of the Java Servlet Specification is that it specifies that servlet containers must allow webapps to specify JSP page(s) that should be precompiled at webapp start time.

For example, if you want index.jsp (in the root of your webapp’s directory) to always be precompiled at webapp startup time, you can add a <servlet> tag for this file in your web.xml file, like this:

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

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

*<servlet>*

*<servlet-name>index.jsp</servlet-name>*

*<jsp-file>/index.jsp</jsp-file>*

*<load-on-startup>0</load-on-startup>*

*</servlet>*

*</web-app>*

Then, Tomcat will automatically precompile index.jsp for you at webapp start time, and the very first request to /index.jsp will be mapped to the precompiled servlet class file of the JSP.

Configuring precompilation in your webapp this way means that all compilation of the JSPs is done

at webapp start time, whether the JSPs are being requested by web clients or not. Each JSP page you declare this way in web.xml will be precompiled. One drawback to this approach is that webapp startup time is then always longer because every page you specify must be precompiledbefore the webapp is accessible to web clients.

Also, the *<load-on-startup>* container tag should contain a positive integer value. This is a loose way to specify precompilation order. The lower you set this number on a JSP page, the earlier in the startup process it will be precompiled.

Precompiling your JSPs in this manner may make your JSPs appear faster to the first web client to request each JSP page after a webapp (re)deployment, however, JSPs that are compiled at build time (before deployment) run slightly faster on every request, even after the first request to each JSP page.

**Precompiling JSPs at build time using JspC**

Here are some valid (as of the time of this writing) reasons for doing build-time pre- compilation of JSPs:

* You need all the performance you can squeeze out of your webapp, and build time compiled JSPs run faster than JSPs that are compiled inside Tomcat after the webapp is deployed. First, the Java class bytecodes generated in both situations should really be the same, and if they’re not exactly the same, the difference will be very small—certainly not worth a major deployment change such as is necessary to precompile the JSPs before deployment. Also, the time it takes Tomcat to compile the original JSP is usually small and occurs only on the first request of each JSP page after webapp deployment/redeployment. All other requests to the JSP pages serve from the compiled and loaded JSP servlet class (JSPs are compiled into Java servlets). But since JSPs that were compiled before webapp deployment are mapped to the URI space in the web.xml file, Tomcat is able to route requests to them slightly faster than if the JSP page were compiled at webapp run- time. This is because when JSP pages are compiled during runtime, the resulting servlets must be mapped to the URI space first by the regular URI mapper, which sends the request to the JspServlet, then the request is mapped to the requested JSP page by Tomcat’s JspServlet. Note that the runtime compiled JSPs are mapped via two layers of indirection (two distinct mappers), and precompiled JSPs

are mapped via only the first layer of indirection. The performance difference comes down to the performance of the two different URI mapper situations. In the end, precompiled JSPs usually run about 4 percent faster. Precompiling them before webapp deployment would save you the small initial request compile time for each JSP page in your webapp, plus the 4 percent performance improvement on each subsequent request for a JSP page. In Tomcat 4.1.x, the runtime JSP request mapper was noticeably slower than the web.xml servlet map- per and made it worth precompiling JSPs before webapp deployment. That made JSP pages faster by approximately 12 percent or so in our tests. But, for Tomcat version 5.0.x and higher, this margin was reduced to about 4 percent or less.

* By precompiling JSPs at webapp build or packaging time, the syntax for the JSPs is checked during the JSP compilation process, which means that you can be confident that the JSPs at least compile with no syntax errors before you deploy your webapp. This is great a way to avoid the situation where you have deployed your webapp to your production server(s) only to find out later that one of the JSPs had a syntax error, and it was found by the first user who requested that page. Also, finding errors in the development phase of the code allows the developer to find and fix the errors more rapidly; it shortens the development cycle. This will not prevent every kind of bug because a compiled JSP may still have runtime logic bugs, but at least you can catch all syntax errors in the development environment.
* If you have a large number of JSP files in your webapp, each of which is some- what long (hopefully you are not copying and pasting lots of content from one JSP page to many other JSP pages; you should instead make use of the JSP include feature), the initial compilation time for all the JSP pages combined could be significantly large. If so, you can save time on the production server by precompiling the JSPs before webapp deployment time. This is especially help- ful if your traffic load is high, and your server responses would otherwise slow down quite a bit, while the server is initially compiling many JSP pages at the same time when the webapp is first started.
* If you have a low server resource situation, for instance, if the Java VM is configured to use a small amount of RAM or the server does not have very many CPU cycles for Tomcat to use, you may not want to do any JSP compilation at all on the server. Instead, you could do the compilation in your development environment and deploy only compiled servlets, which would lighten the utilization of both memory and CPU time for the first request of each JSP file after each new copy of the webapp is deployed.
* You are developing a JSP web application that you will sell to customer(s) whom you do not want to have the JSP source code. If you could give the customer(s) the webapp containing just compiled servlets, you could develop the webapp using the original JSPs, and ship it with the compiled JSP servlets. In this use case, pre- compiling before release to the customer is used as a source code obfuscation mechanism. Keep in mind, though, that compiled Java class files are relatively easy to decompile into readable Java source code, but (as of this writing) there is no way to decompile it all the way back into JSP source code.
* Also, as of Tomcat version 5.5, you no longer needa JDK that has a built-in Java source compiler to serve runtime compiled JSPs. Tomcat versions 5.5 and higher come bundled with the Eclipse JDT compiler, which is a Java compiler that is itself written in pure Java. Because the JDT compiler is bundled as part of Tom- cat, Tomcat can always compile JSPs into servlets, even when Tomcat is run on a JRE and not a JDK.

[Example 10-4](#_bookmark792) is an Ant buildfile that you can use to compile your webapp’s JSP files at build time.

*Example 10-4. The precompile-jsps.xml Ant build file*

*<project name="pre-compile-jsps" default="compile-jsp-servlets">*

*<!-- Private properties. -->*

*<property name="webapp.dir" value="${basedir}/webapp-dir"/>*

*<property name="tomcat.home" value="/opt/tomcat"/>*

*<property name="jspc.pkg.prefix" value="com.mycompany"/>*

*<property name="jspc.dir.prefix" value="com/mycompany"/>*

*<!-- Compilation properties. -->*

*<property name="debug" value="on"/>*

*<property name="debuglevel" value="lines,vars,source"/>*

*<property name="deprecation" value="on"/>*

*<property name="encoding" value="ISO-8859-1"/>*

*<property name="optimize" value="off"/>*

*<property name="build.compiler" value="modern"/>*

*<property name="source.version" value="1.5"/>*

*<!-- Initialize Paths. -->*

*<path id="jspc.classpath">*

*<fileset dir="${tomcat.home}/bin">*

*<include name="\*.jar"/>*

*</fileset>*

*<fileset dir="${tomcat.home}/server/lib">*

*<include name="\*.jar"/>*

*</fileset>*

*<fileset dir="${tomcat.home}/common/i18n">*

*<include name="\*.jar"/>*

*</fileset>*

*<fileset dir="${tomcat.home}/common/lib">*

*<include name="\*.jar"/>*

*</fileset>*

*<fileset dir="${webapp.dir}/WEB-INF">*

*<include name="lib/\*.jar"/>*

*</fileset>*

*<pathelement location="${webapp.dir}/WEB-INF/classes"/>*

*<pathelement location="${ant.home}/lib/ant.jar"/>*

*<pathelement location="${java.home}/../lib/tools.jar"/>*

*</path>*

*<property name="jspc.classpath" refid="jspc.classpath"/>*

*<!-- ========================================================== -->*

*<!-- Generates Java source and a web.xml file from JSP files. -->*

*<!-- ========================================================== -->*

*<target name="generate-jsp-java-src">*

*<mkdir dir="${webapp.dir}/WEB-INF/jspc-src/${jspc.dir.prefix}"/>*

*<taskdef classname="org.apache.jasper.JspC" name="jasper2">*

*<classpath>*

*<path refid="jspc.classpath"/>*

*</classpath>*

*</taskdef>*

*<touch file="${webapp.dir}/WEB-INF/jspc-web.xml"/>*

*<jasper2 uriroot="${webapp.dir}" package="${jspc.pkg.prefix}" webXmlFragment="${webapp.dir}/WEB-INF/jspc-web.xml"*

*outputDir="${webapp.dir}/WEB-INF/jspc-src/${jspc.dir.prefix}" verbose="1"/>*

*</target>*

*<!-- ========================================================== -->*

*<!-- Compiles (generates Java class files from) the JSP servlet -->*

*<!-- source code that was generated by the JspC task. -->*

*<!-- ========================================================== -->*

*<target name="compile-jsp-servlets" depends="generate-jsp-java-src">*

*<mkdir dir="${webapp.dir}/WEB-INF/classes"/>*

*<javac srcdir="${webapp.dir}/WEB-INF/jspc-src" destdir="${webapp.dir}/WEB-INF/classes" includes="\*\*/\*.java"*

*debug="${debug}" debuglevel="${debuglevel}" deprecation="${deprecation}" encoding="${encoding}" optimize="${optimize}" source="${source.version}">*

*<classpath>*

*<path refid="jspc.classpath"/>*

*</classpath>*

*</javac>*

*</target>*

*<!-- ========================================================= -->*

*<!-- Cleans any pre-compiled JSP source, classes, jspc-web.xml -->*

*<!-- ========================================================= -->*

*<target name="clean">*

*<delete dir="${webapp.dir}/WEB-INF/jspc-src"/>*

*<delete dir="${webapp.dir}/WEB-INF/classes/${jspc.dir.prefix}"/>*

*<delete file="${webapp.dir}/WEB-INF/jspc-web.xml"/>*

*</target>*

*</project>*

If you put this Ant build xml content into a file named something such as pre- compile-jsps.xml, you can test it alongside any build.xml file you already have, and if you like it, you can merge it into your build.xml.

This build file will find all of your webapp’s JSP files, compile them into servlet classes, and generate servlet mappings for those JSP servlet classes. The servlet mappings it generates must go into your webapp’s WEB-INF/web.xml file, but it would be difficult to write an Ant build file that knows how to insert the servlet mappings into your web.xml file in a repeatable way every time the buildfile runs. Instead, we used an XML entity include so that the generated servlet mappings go into a new file every time the buildfile runs and that servlet mappings file can be inserted into your web.xml file via the XML entity include mechanism. To use it, your webapp’s WEB- INF/web.xml must have a special entity declaration at the top of the file, plus a reference to the entity in the content of the web.xml file where you want the servlet map- pings file to be included. Here is how an empty servlet 2.5 webapp’s web.xml file looks with these modifications:

*<!DOCTYPE jspc-webxml [*

*<!ENTITY jspc-webxml SYSTEM "jspc-web.xml">*

*]>*

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

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

*<!-- We include the JspC-generated mappings here. -->*

*&jspc-webxml;*

*<!-- Non-generated web.xml content goes here. -->*

*</web-app>*

Make sure your webapp’s web.xml file has the inline DTD (the DOCTYPE tag) all the way at the top of the file andthe servlet 2.5 web-app schema declaration below that. Then, wherever you want to insert the generatedservlet mappings in your web.xml file, put the entity reference &jspc-webxml;. Remember, the entity reference begins with an ampersand (&), then has the name of the entity, and ends with a semicolon (;).

To use the buildfile, just edit it and set all of the properties at the top to values that match your setup, and then run it like this:

$ **ant -f pre-compile-jsps.xml**

Buildfile: pre-compile-jsps.xml

generate-jsp-java-src:

[jasper2] Sep 27, 2008 10:47:15 PM org.apache.jasper.xmlparser.MyEntityResolver resolveEntity

[jasper2] SEVERE: Invalid PUBLIC ID: null

[jasper2] Sep 27, 2007 10:47:17 PM org.apache.jasper.JspC processFile [jasper2] INFO: Built File: /index.jsp

compile-jsp-servlets:

[javac] Compiling 1 source file to /home/jasonb/myproject/webapp-dir/WEB-INF/ classes

BUILD SUCCESSFUL

Total time: 7 seconds

Any JSP files you have in your webapp dir will be compiled into servlets, and when you deploy the webapp, the JSP page requests will be mapped to the compiled servlets. Ignore the “SEVERE: Invalid PUBLIC ID: null” message if you get it; it’s bogus. If you want to clean out the compiled servlets and their generated Java source and mappings, just execute the clean target like this:

*$ ant -f pre-compile-jsps.xml clean*

One thing that this build file does not do: remove all of the JSP files in your webapp after compiling them. We didn’t want you to accidentally delete your JSP files, so we intentionally left it out. Your own build file should do that before the webapp gets deployed. If you forget and accidentally leave the JSP files in the deployed webapp, none of them should get served by Tomcat because the web.xml file explicitly tells Tomcat to use the compiled servlet classes instead.

**Capacity Planning**

Capacity planning is another important part of tuning the performance of your Tomcat server in production. Regardless of how much configuration file-tuning and testing you do, it won’t really help if you don’t have the hardware and bandwidth your site needs to serve the volume of traffic you are expecting.

Here’s a loose definition of capacity planning as it fits into the context of this section: capacity planning is the activity of estimating the necessary computer hard- ware, operating system, and bandwidth necessary for a web site by studying and/or estimating the total network traffic a site will have to handle, deciding on acceptable service characteristics, and finding appropriate hardware and operating systems that meet or exceed the server software’s requirements to meet the service requirements. In this case, the server software includes Tomcat, as well as any third-party web servers and load balancers that you are using “in front” of Tomcat.

If you don’t do any capacity planning before you buy and deploy your production servers, you won’t know if the server hardware can handle your web site’s traffic load. Or, worse still, you won’t realize the error until you’ve already ordered, paid for, and deployed applications on the hardware—usually too late to change direction very much. You can usually add a larger hard drive or even order more server computers, but sometimes it’s less expensive overall to buy and/or maintain fewer server computers in the first place.

The higher the volume of traffic on your web site, or the larger the load that is generated per client request, the more important capacity planning becomes. Some sites get so much traffic that only a cluster of server computers can handle it all within reasonable response time limits. Conversely, sites with less traffic have less of a problem finding hardware that meets all their requirements. It’s true that throwing more or bigger hardware at the problem usually fixes things, but, especially in the high traffic cases, that may be prohibitively costly. For most companies, the lower the hardware costs are (including ongoing maintenance costs after the initial purchase), the higher profits can be. Another factor to consider is employee productivity. If having faster hardware would make the developers 20 percent more effective in getting their work done quickly, for example, then depending on the size of the team, it may be worth the hardware cost difference to order bigger/faster hardware up front.

Capacity planning is usually done at upgrade points as well. Before ordering replacement hardware for

existing mission-critical server computers, it’s probably a good idea to gather information about what your company needs, based on updated requirements, common traffic load, software footprints, etc.

There are at least a couple of common methods of arriving at decisions when con- ducted capacity planning. In practice, we’ve seen two main types: anecdotal approaches and academic approaches, such as enterprise capacity planning.

**Anecdotal Capacity Planning**

Anecdotal capacity planning is a sort of light capacity planning that isn’t meant to be exact, but close enough to keep a company out of situations that would be caused by doing no capacity planning at all. This method follows capacity and performance trends that are obtained from previous industry experience. For example, you could make your best educated guess at how much outgoing network traffic your site will have at its peak usage (hopefully from some other real-world site), and double that figure. That figure is your site’s new outgoing bandwidth requirement for which you will make sure to buy and deploy hardware that can handle it. Most people will do capacity planning this way because it’s quick and requires little effort and time.

**Enterprise Capacity Planning**

Enterprise capacity planning is meant to be more exact and takes much longer. This method is necessary for sites with a very high volume of traffic, often combined with a high load per request. Detailed capacity planning like this is necessary to keep hard- ware and bandwidth costs as low as they can be, while still providing the quality of service that the company guarantees or is contractually obligated to live up to. Usually, this involves the use of commercial capacity planning analysis software in addition to iterative testing and modeling. Few companies do this kind of capacity planning, but the few that do are very large enterprises that have a budget large enough to afford doing it (mainly because this sort of thorough planning ends up paying for itself).

The biggest difference between anecdotal and enterprise capacity planning is depth. Anecdotal capacity planning is governed by rules of thumb and is more of an educated guess, whereas enterprise capacity planning is an in-depth requirements-and- performance study whose goal is to arrive at numbers that are as exact as possible.

**Capacity Planning on Tomcat**

To capacity plan for server machines that run Tomcat, you could study and plan for any of the following items (this isn’t meant to be a comprehensive list, but instead a list of some common items):

***Server computer hardware***

Which computer architecture(s)? How many computers will your site need? One big one? Many smaller ones? How many CPUs per computer? How much RAM? How much hard drive space and what speedI/O? What will the ongoing maintenance be like? How does switching to different JVM implementations affect the hardware requirements?

***Network bandwidth***

How much incoming and outgoing bandwidth will be needed at peak times? How might the web application be modified to lower these requirements?

***Server operating system***

Which operating system works best for the job of serving your site? Which JVM implementations are available for each operating system, and how well does each one take advantage of the operating system? For example, does the JVM support native multithreading? Symmetric multiprocessing (SMP)? If SMP is supported by the JVM, should you consider multiprocessor server computer hardware? Which serves your webapp faster, more reliably, and less expensively: multiple single- processor server computers or a single four-CPU server computer?

Here’s a general procedure for all types of capacity planning, and one that is particularly applicable to Tomcat:

1. Characterize the workload. If your site is already up and running, you can mea- sure the requests per second, summarize the different kinds of possible requests, and measure the resource utilization per request type. If your site isn’t running yet, you can make some educated guesses at the request volume and run staging tests to determine the resource requirements.
2. Analyze performance trends. You need to know what requests generate the most load and how other requests are in comparison. Knowing which requests generate the most load or use the most resources, will help you know what to optimize to have the best overall impact on your server computers. For example, if a servlet that queries a database takes too long to send its response, maybe caching some of the data in RAM would safely improve response time.
3. Decide on minimum acceptable service requirements. For example, you may not want the end user to ever wait longer than 20 seconds for a web page response. That means that even during peak load, no request’s total time from the initial request to the completion of the response can take longer than 20 seconds. That may include any and all database queries and filesystem access needed to complete the heaviest resource-intensive request in your application. The minimum acceptable service requirements are up to each company and vary from company to company. Other kinds of service minimums include the number of requests per second the site must be able to serve and the minimum number of concurrent sessions and users.
4. Decide what infrastructure resources you will use, and test it in a staging environment. Infrastructure resources include computer hardware, bandwidth circuits, operating system software, and so on. Order, deploy, and test at least one server machine that mirrors what you’ll have for production and see if it meets your requirements. While testing Tomcat, make sure you try more than one JVM implementation, try different memory size settings, and request thread pool sizes (discussed earlier in this chapter).
5. If step 4 meets your service requirements, you can order and deploy more of the same thing to use as your production server computers. Otherwise, redo step 4 until service requirements are met.

Be sure to document your work because it tends to be a time-consuming process that must be repeated if someone needs to know how your company arrived at the answers. Also, because the testing is an iterative process, it’s important to document all of the test results on each iteration and the configuration settings that produced the results so you know when your tuning is no longer yielding noticeable positive results.

Once you’ve finished with your capacity planning, your site will be much better tuned for performance, mainly due to the rigorous testing of a variety of options. You should have gained a noticeable amount of performance just by having the right hardware, operating system, and JVM combination for your particular use of Tomcat.