TomEE Performance Tuning

Once you have TomEE up and running, you will likely want to do some performance tuning so that it serves requests more efficiently on your computer.

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 TomEE 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 bottlenecks 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 TomEE runs on, you will skew your test results, sometimes badly. Ideally, you should run TomEE 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 TomEE 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 TomEE 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 TomEE logfiles for error messages.

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. 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 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), Siege (see http://www.joedog.org/joedog/siege), and JMeter from Apache Jakarta (see 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 your TomEE 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 preinstalled in most operating systems, but its build and install instructions are straightforward 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 TomEE 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 support 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.

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 4-1 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 carefully. 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 and the more likely the benchmark results will show how TomEE performs overall. You should benchmark 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 TomEE'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 TomEE 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 4-1. Benchmarking with ab

```
$ **ab -k -n 100000 -c 149 http://*tomEEhost*:8080**
This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-
2.0
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
http://www.zeustech.net/
Copyright 1997-2005 The Apache Software Foundation,
http://www.apache.org/

Benchmarking tomEEhost (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:        tomEEhost
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 TomEE, which is less efficient because it must connect a new TCP socket to TomEE to make each HTTP request. The result is that fewer requests per second will be handled, and the throughput from TomEE to the client (*ab*) will be smaller (see Example 4-2).

Example 4-2. Benchmarking with ab with keep-alive connections disabled

```
$ ab -n 100000 -c 149 http://tomEEhost:8080/
This is ApacheBench, Version 2.0.40-dev <$Revision$> apache-
2.0
```

```
Copyright 1996 Adam Twiss, Zeus Technology Ltd,
http://www.zeustech.net/
Copyright 1997-2005 The Apache Software Foundation,
http://www.apache.org/

Benchmarking tomEEhost (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:        tomEEhost
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 4-3 shows the *siege* command line and the results from the benchmark test.

Example 4-3. Benchmarking with siege with keep-alive connections disabled

```
$ siege -b -r 671 -c 149 tomEEhost: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,[23] and all 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*.
- The throughput reported by *siege* is significantly lower than that reported by *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 TomEE, 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 need to perform garbage collection. If you have enough memory in the machine on which you're running JMeter, you should change both of the `256` numbers to something higher, such as `512`. It is important to do this first because this setting'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
```

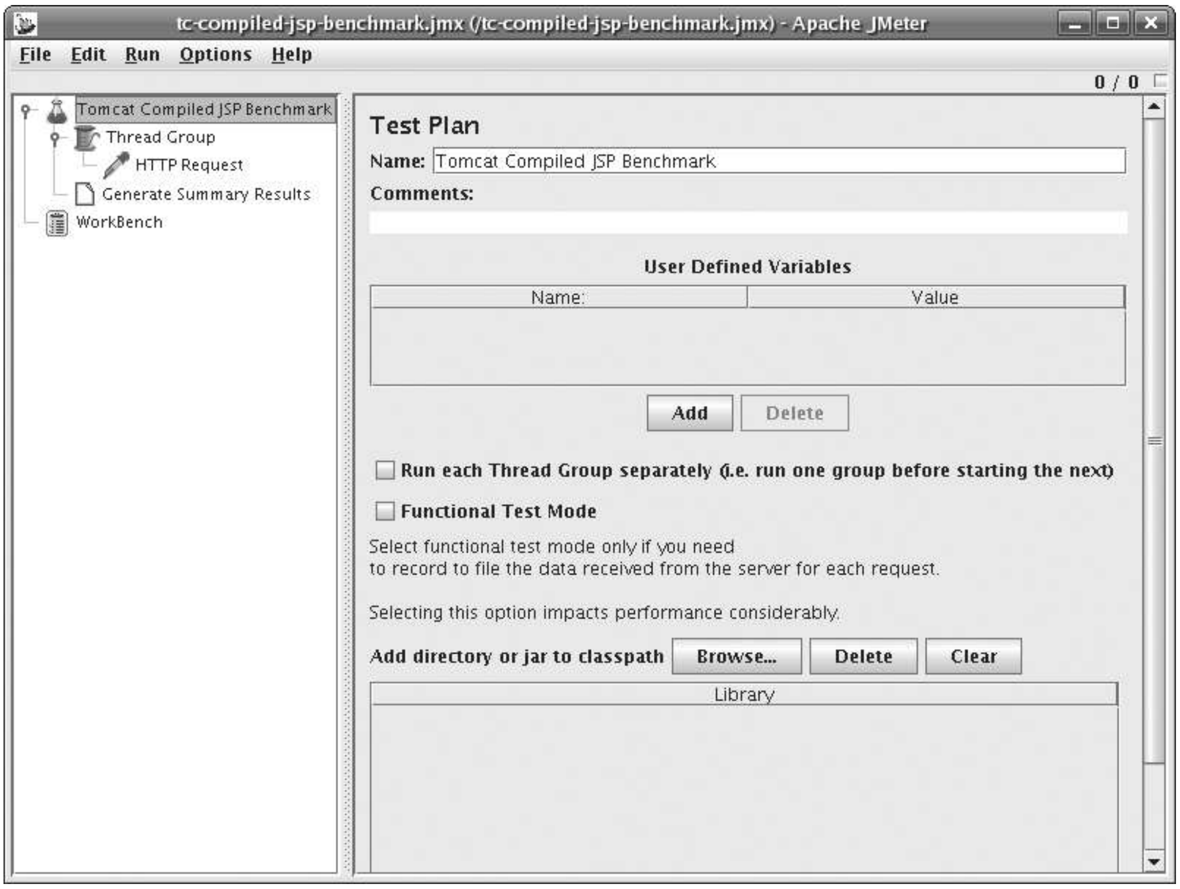
JMeter's screen is laid out as a tree view on the left and a selection details panel on the right. Select something in the tree view and you can see the details of that item in the details panel on the right. To run any tests, you must assemble and configure the proper objects in the tree, and then JMeter 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`.
3. Right click on the `Thread Group` tree node and select `Add` → `Sampler` → `HTTP Request`.
4. In the HTTP request details panel, change the `Web Server` settings to point to your TomEE server and its port number, and change the `Path` under the `HTTP Request` settings to the URI in

your TomEE installation that you would like to benchmark. For instance `/`.
5. Right click on the `Thread Group` tree node again and select `Add` → `Post Processors` → `Generate Summary Results`.
6. 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 4-1 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.


tc-compiled-jsp-benchmark.jmx (/tc-compiled-jsp-benchmark.jmx) - Apache JMeter

File Edit Run Options Help

0 / 0

- Tomcat Compiled JSP Benchmark
  - Thread Group
    - HTTP Request
  - Generate Summary Results
- WorkBench

Test Plan

Name: Tomcat Compiled JSP Benchmark

Comments:

User Defined Variables

Name:	Value

Add Delete

☐ Run each Thread Group separately (i.e. run one group before starting the next)

☐ Functional Test Mode

Select functional test mode only if you need
to record to file the data received from the server for each request.

Selecting this option impacts performance considerably.

Add directory or jar to classpath Browse... Delete Clear

Library

Figure 4-1. Apache JMeter GUI showing the fully assembled test plan

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 TomEE, 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 TomEE 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 TomEE's server code.

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` → `Graph Results`.
4. Save your test plan under a new name; this time for graphical viewing of test results.

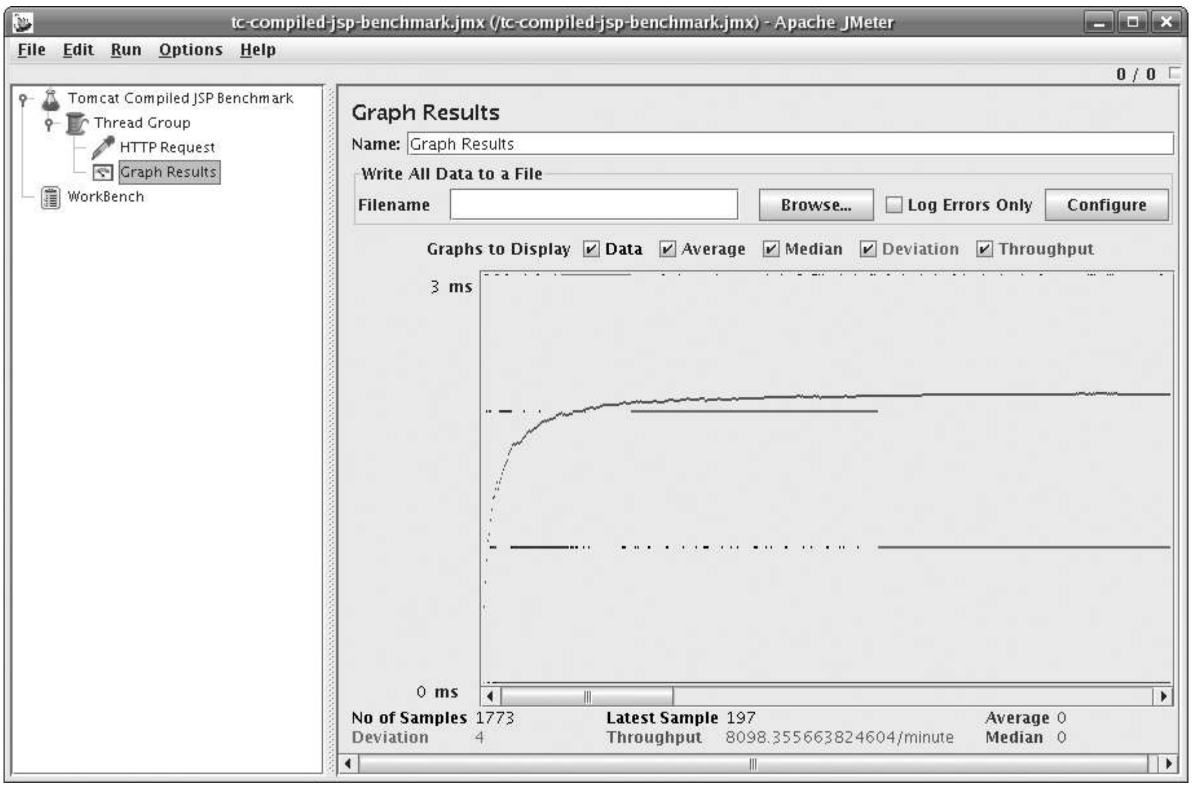
5. Select the `Graph Results` tree node.

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

Tip

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 4-2 shows the JMeter GUI graphing the test results.


tc-compiled-jsp-benchmark.jmx (/tc-compiled-jsp-benchmark.jmx) - Apache JMeter
File Edit Run Options Help
0 / 0
Tomcat Compiled JSP Benchmark
Thread Group
HTTP Request
Graph Results
WorkBench
Graph Results
Write All Data to a File
Browse...
Configure
3 ms
0 ms

Name: Graph Results

Filename

Log Errors Only [ ]

Graphs to Display [✓ Data] [✓ Average] [✓ Median] [✓ Deviation] [✓ Throughput]

No of Samples 1773 Latest Sample 197 Average 0
Deviation 4 Throughput 8098.355663824604/minute Median 0

Figure 4-2. 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, plotting 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 command line. 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 command line, and then 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 command line. Use the results you obtain on the command line 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 TomEE that demonstrates a benchmark procedure from start to finish and also yields some information that can help you configure TomEE so that it performs better for your web application.

We benchmarked all of TomEE's web server implementations, plus Apache *httpd* standalone, plus Apache *httpd*'s modules that connect to TomEE to see how fast each configuration is at serving static content. For example, is Apache *httpd* faster than TomEE standalone? Which TomEE 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 configurations, at least for one hardware, OS, and Java combination.

Tip

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 hardware 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 TomEE and/or Apache *httpd* will perform on your machine is to benchmark it yourself following a similar benchmark test procedure.

TomEE connectors and Apache httpd connector modules

TomEE 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 TomEE's default connector implementation, unless the APR `Connector`'s *libtcnative* library is found at TomEE 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. Is it? We'll find out.

APR (Apache Portable Runtime)

This is TomEE's default connector implementation if you install TomEE on Windows via the NSIS installer, but it is not the default connector implementation for most other stock installations of TomEE. 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 TomEE 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 TomEE'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 nonblocking TCP socket features. The main goal of this `Connector` design is to offer TomEE administrators a `Connector` implementation that performs better than the JIO `Connector` by using fewer threads by implementing parts of the `Connector` in a nonblocking 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 TomEE `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 TomEE 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 TomEE project. It began years before Apache *httpd*'s *mod_proxy* included support for the AJP protocol (TomEE'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 TomEE 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 TomEE via TCP to TomEE's AJP server port, sends requests through to TomEE, waits for TomEE's responses, and then Apache *httpd* forwards the responses to the web client(s). The requests go through Apache *httpd* to TomEE and back, and the protocol used between Apache *httpd* and TomEE 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 and is 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 TomEE via TCP, but this time it connects to TomEE'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 TomEE's web server, TomEE responds, and *httpd* forwards the response to the web client. All communication between Apache *httpd* and TomEE 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

We chose two different kinds of server hardware to benchmark running the server software. Here are descriptions of the two types of computers on which we ran the benchmarks:

Desktop: Dual Intel Xeon 64 2.8Ghz CPU, 4G RAM, SATA 160G HD 7200RPM

This was a tower machine with two Intel 64-bit CPUs; each CPU was single core and hyperthreaded.

Laptop: AMD Turion64 ML-40 2.2Ghz CPU, 2G RAM, IDE 80G HD 5400RPM

This was a laptop that has a single 64-bit AMD processor (single core).

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) around the 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 wanted to benchmark and that the client was fast enough to give us the most accurate results. 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.

TomEE's JVM startup switch settings were:

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

Here is our TomEE 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 TomEE JVM process's environment, and then restarting TomEE.

We built the APR connector like this:

```
# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse
-mmmx -msse -msse2 -mss

e3 -m3dnow -mtune=athlon64" ./configure --with-
apr=/usr/bin/apr-1-config

--prefix=/opt/

tomEE/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 tomEE-connectors-1.2.20-src/native

# CFLAGS="-O3 -falign-functions=0 -march=athlon64 -mfpmath=sse
-mmmx -msse -msse2 -mss

e3 -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 and built it two different ways and benchmarked each of the resulting binaries. We built it for prefork 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 need to log anything per each request (just as we configured TomEE). 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 benchmark 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 TomEE'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
```

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 TomEE and Apache *httpd* would easily saturate our gigabit ethernet link between the client machine and the server machine. Again, we copied this file into TomEE'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
```

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

1. Configure and restart the Apache *httpd* and/or TomEE 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 TomEE 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 Figure 4-3 and Figure 4-4).

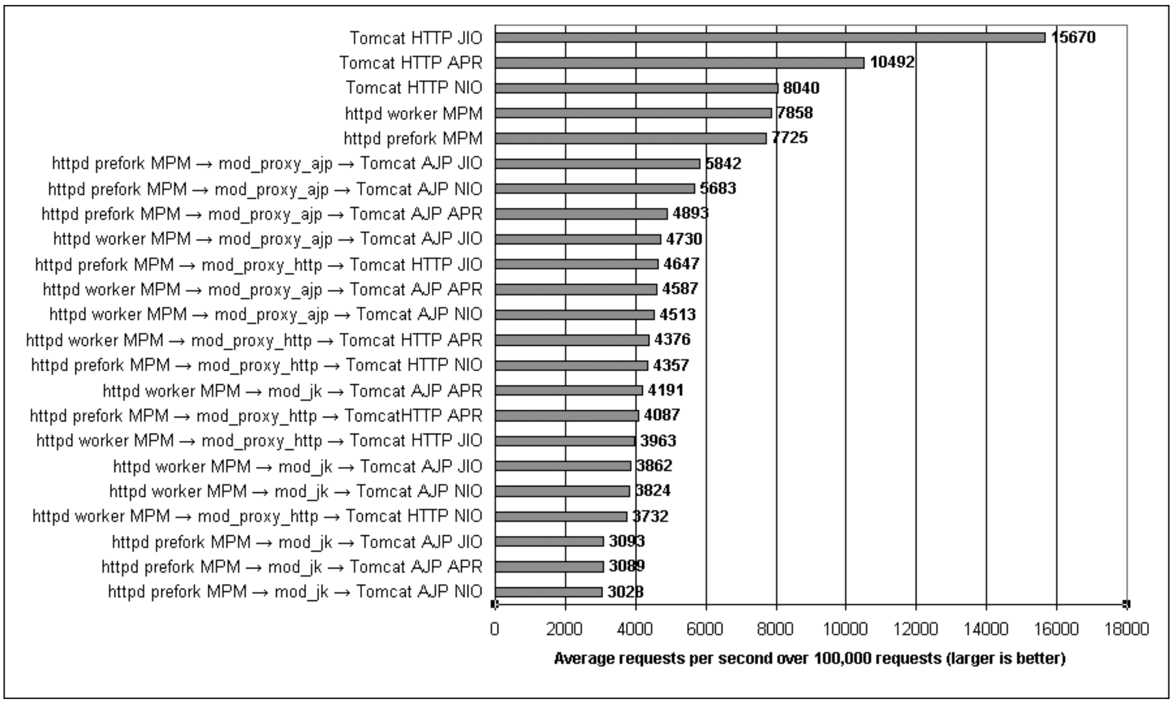


Figure 4-3. Benchmark results for serving small text files on the AMD64 laptop

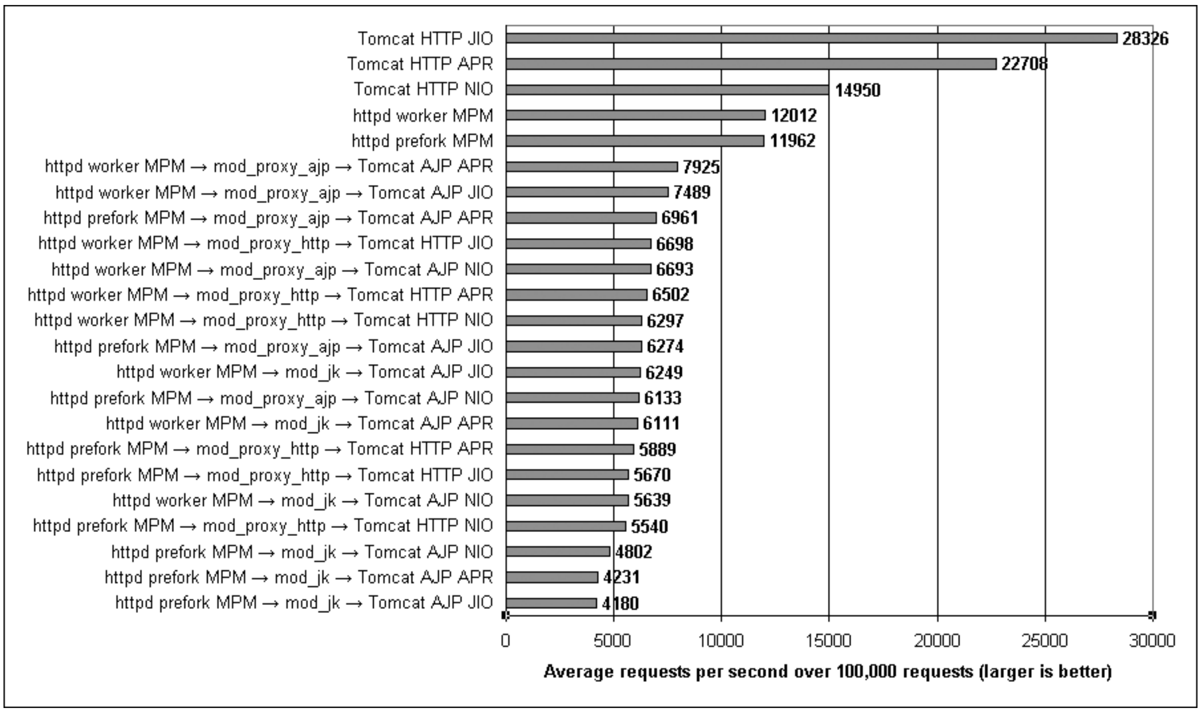


Figure 4-4. Benchmark results for serving small text files on the EM64T tower

Notice that Figure 4-3 and Figure 4-4 look very similar. On both machines, TomEE 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 TomEE via a connector module. And, dominating the slow end of the graphs is *mod_jk*.

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 TomEE and Apache *httpd* are able to take advantage of that.

Next, we examine the results of the 9k image files benchmark on both machines. Figure 4-5 and Figure 4-6 show the results for the AMD64 computer and the dual EM64T computer, respectively.

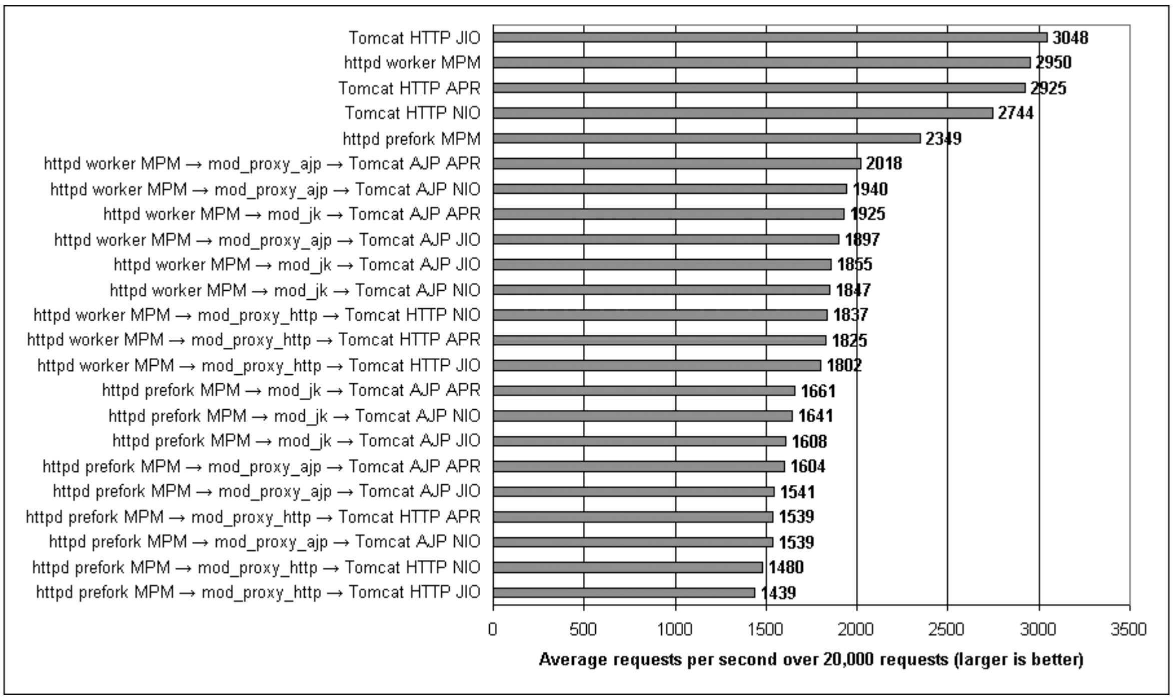


Figure 4-5. Benchmark results for serving 9k image files on the AMD64 laptop

In Figure 4-5, you can see that on AMD64, TomEE standalone JIO wins again, with Apache *httpd* worker MPM trailing close behind. In this benchmark, their performance is nearly identical, with TomEE standalone APR in a very close third place. TomEE standalone NIO is in fourth place, trailing a little behind APR. Apache *httpd* prefork MPM is fifth fastest again behind all of the TomEE standalone configurations. Slower still are all of the permutations of Apache *httpd* connecting to TomEE 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 is somewhat different, showing that on the dual EM64T, Apache *httpd* edges slightly ahead of TomEE 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 TomEE 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 TomEE standalone connectors performing better than Apache *httpd* prefork in this set of benchmarks. The configurations where Apache *httpd* connects to TomEE via a connector module were again the slowest performing configurations, with *mod_jk* performing the slowest.

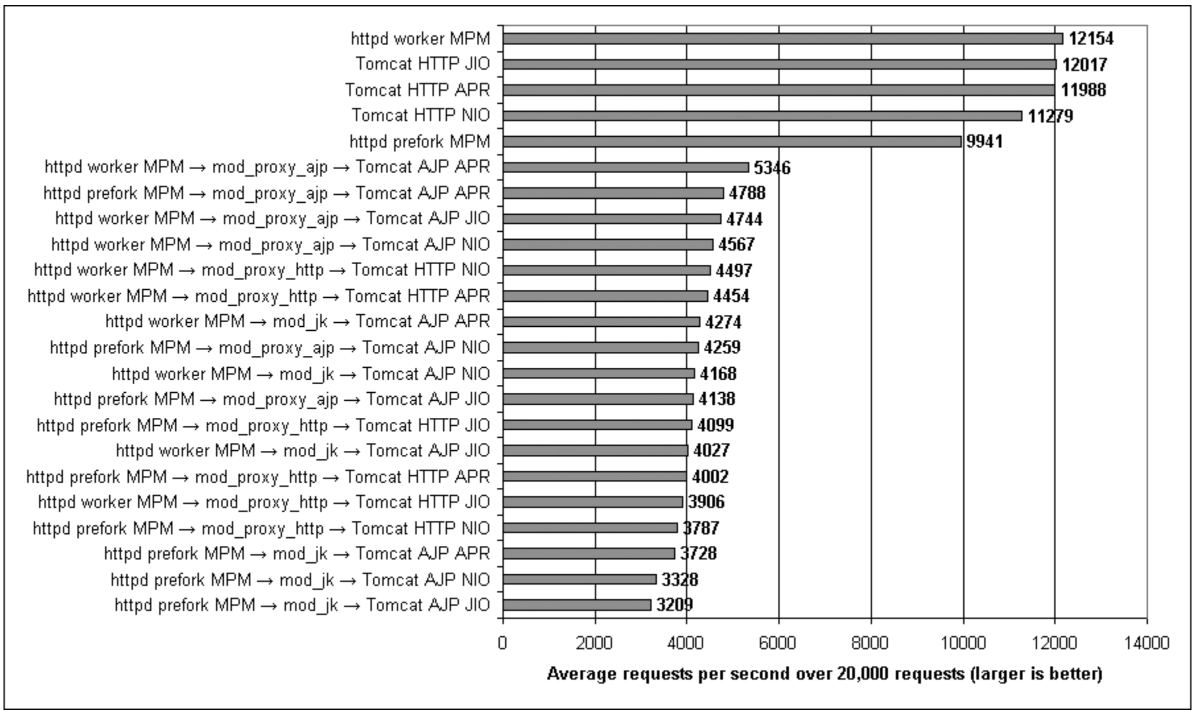


Figure 4-6. Benchmark results for serving 9k image files on the EM64T tower

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 TomEE 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:

- TomEE 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 TomEE. We observed that TomEE standalone JIO was almost always the fastest way to serve static resources. TomEE served them between 3 percent and 136 percent faster than Apache *httpd* in our benchmarks—TomEE 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 perform 1 percent faster than TomEE. But in the small files benchmark, TomEE was a minimum of 99 percent faster than Apache *httpd* and a 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.
- TomEE standalone (configured to use any HTTP connector implementation) was always faster than Apache *httpd* built and configured for prefork MPM; TomEE standalone was a minimum of

21 percent faster than Apache *httpd* and a maximum of 30 percent faster than Apache *httpd* for 9k image files, and for small files TomEE was a minimum of 103 percent faster than Apache *httpd* and a maximum of 136 percent faster than Apache *httpd* prefork MPM.

- Apache *httpd* was quite a bit slower at serving small files. TomEE standalone's JIO, APR, and NIO connectors were each faster than Apache *httpd*—TomEE's JIO connector performed as much as 136 percent faster than Apache *httpd*'s fastest configuration, TomEE's APR connector performed 89 percent faster than Apache *httpd*, and TomEE 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 TomEE standalone's three HTTP connectors.
- Serving TomEE's resources through Apache *httpd* was very slow compared to serving them directly from TomEE. When we compared the benchmark results between TomEE standalone and TomEE serving through Apache *httpd* via *mod_proxy*, TomEE standalone consistently served at least 51 percent faster when using only TomEE'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, TomEE standalone was a minimum of 168 percent faster than the Apache *httpd* to TomEE configurations and a maximum of 578 percent faster! That's not a misprint—it's really 578 percent faster. For the 9k image files benchmark, TomEE 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 TomEE connector design, but it was usually smaller, with *mod_proxy_ajp* averaging about 13 percent faster than *mod_proxy_http*.

- Serving TomEE'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 TomEE) 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 TomEE is, serving its files through Apache *httpd* throttles TomEE down so that TomEE 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 TomEE'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 TomEE to serve static resource faster than Apache *httpd*? The main reason we can think of: because TomEE 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 TomEE enjoys.

Another potential reason TomEE 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 TomEE, 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—TomEE 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 benchmarking 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 TomEE 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 TomEE's web server implementations compares with that of Apache *httpd* in a specific limited environment and for specific tests.