Rev up your Tomcat server on IBM i

Tian Gang
Yang Cheng Peng
Zheng Chang Qing

August 28, 2014

Performance and security are two important key points when running Tomcat on IBM i. This article introduces how to improve Tomcat’s performance and security by tuning IBM i, Java virtual machine (JVM), Tomcat, HTTP Server for i and specific applications.

Introduction

This is the second article in a two-part series focused on IBM i Tomcat enablement. The first article discussed how to put Tomcat in IBM i, associate it with the HTTP Server for i and run web applications in a Tomcat-based web environment. As a foundation building article, it is strongly suggested that you read and follow the processes outlined in the first article before continuing with this article.

Part two (this article) explains the best practices for running IBM i web solutions on *Apache Software Foundation (*ASF) Tomcat. After having a basic understanding of how to install and run Tomcat on IBM i, it is time to pay attention to Tomcat's performance capabilities and controls to enable Tomcat to achieve optimum performance on IBM i.

This article uses a sample application TomcatTestServlets, revisited from Part 1 and revised to demonstrate how to improve performance of the Tomcat server by using or tuning the following: IBM i subsystems, JVM, Tomcat, HTTP Server for i, and the web application itself.

Here are a few preliminary things to keep in mind when tuning your application server:

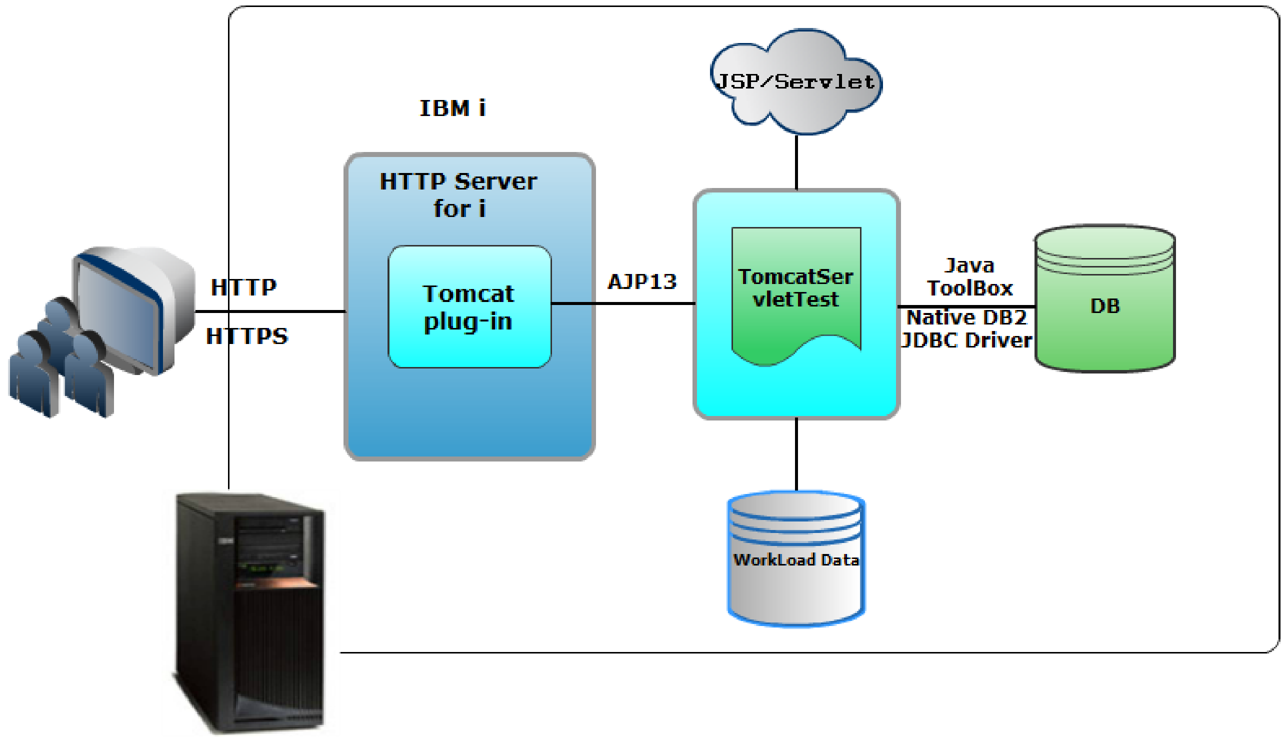
- Several factors besides the application server can impact performance, including hardware, operating system configuration, conflicting demand for system resources, performance of back-end database resources, network latency, and so on. You must factor in these items when conducting performance evaluations.
- The sample application is just a demo and there is no guarantee that all the tuning parameters are perfectly suited for your applications. As a result, it is important for you to conduct focused performance testing and tuning against your own applications.
- Performance improvement may involve sacrificing a certain level of feature or function in the application or the application server. The tradeoff between performance and feature must be

weighed carefully when evaluating performance tuning changes. For example, when tuning the performance of HTTP Server for i, the function of Domain Name Server (DNS) looking up is disabled to speed up transmission.
- This article focuses on the principles of tuning application performance. While each application has its own unique characteristics, the principles apply to most of the tuning efforts.

Overview of a sample application

The TomcatTestServlets sample application is used to test the throughput for a database query using the IBM DB2® JDBC driver that is provided by the IBM® Toolbox for Java. We updated this application to make it not only serve as an excellent application for database functional testing, but also to provide a standard set of workloads for measuring application server performance. Refer to the application's TomcatTestServlets.war file (that can be extracted from the TomcatTestServlets.zip file) in the Downloadable resources section. Figure 1 shows a high-level overview of the application's architecture.

Figure 1. Overview of the sample application using a Tomcat application server



Establish a baseline

Before any improvements can be made, it is essential to establish a baseline of the application's current performance. This is important so that any changes can be measured in a useful and quantifiable manner. There are endless metrics to consider measuring when tuning performance, but **application throughput** and **response time** are two of the more important.

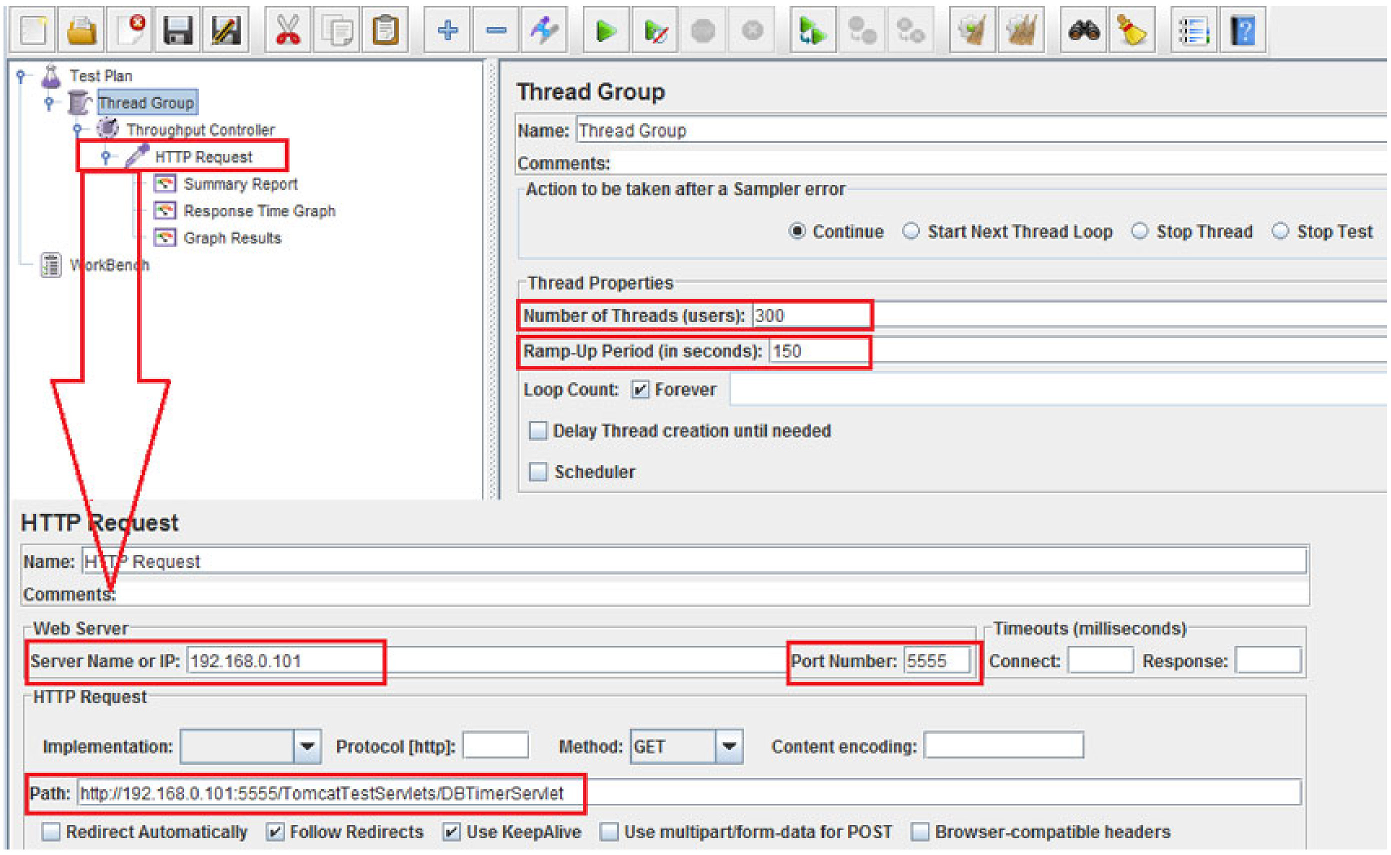
In order to test the web application's performance, an easy-to-use testing tool is needed. There are a wide variety of testing tools you can use. Some of the most commonly used tools are:

- Apache JMeter – A free testing tool designed by Apache to load test functional behavior and measure performance
- Apache HTTP server benchmarking tool – A free testing tool designed by Apache for benchmarking your Apache Hypertext Transfer Protocol (HTTP) server
- IBM Rational Performance Tester (RPT) – A commercial performance testing tool developed by IBM

The Apache tools can be obtained for free from the Apache website.

This article uses RPT as the testing tool, but a brief introduction is provided for Apache JMeter. Apache JMeter is a 100% pure Java application designed to drive and measure the *load test* functional behavior. It can be used to simulate a heavy load on a server, network, or object to test its strength or to analyze the overall performance under different load types. With Apache JMeter, you can easily create a performance test as shown in Figure 2.

Figure 2. Create performance test using Apache JMeter



Thread Group

Name: Thread Group

Comments:

Action to be taken after a Sampler error: [● Continue] ○ Start Next Thread Loop ○ Stop Thread ○ Stop Test

Thread Properties

Number of Threads (users): 300

Ramp-Up Period (in seconds): 150

Loop Count: [✓ Forever]

☐ Delay Thread creation until needed

☐ Scheduler

HTTP Request

Name: HTTP Request

Comments:

Web Server

Server Name or IP: 192.168.0.101

Port Number: 5555

Timeouts (milliseconds) Connect: Response:

HTTP Request

Implementation: Protocol [http]: Method: GET Content encoding:

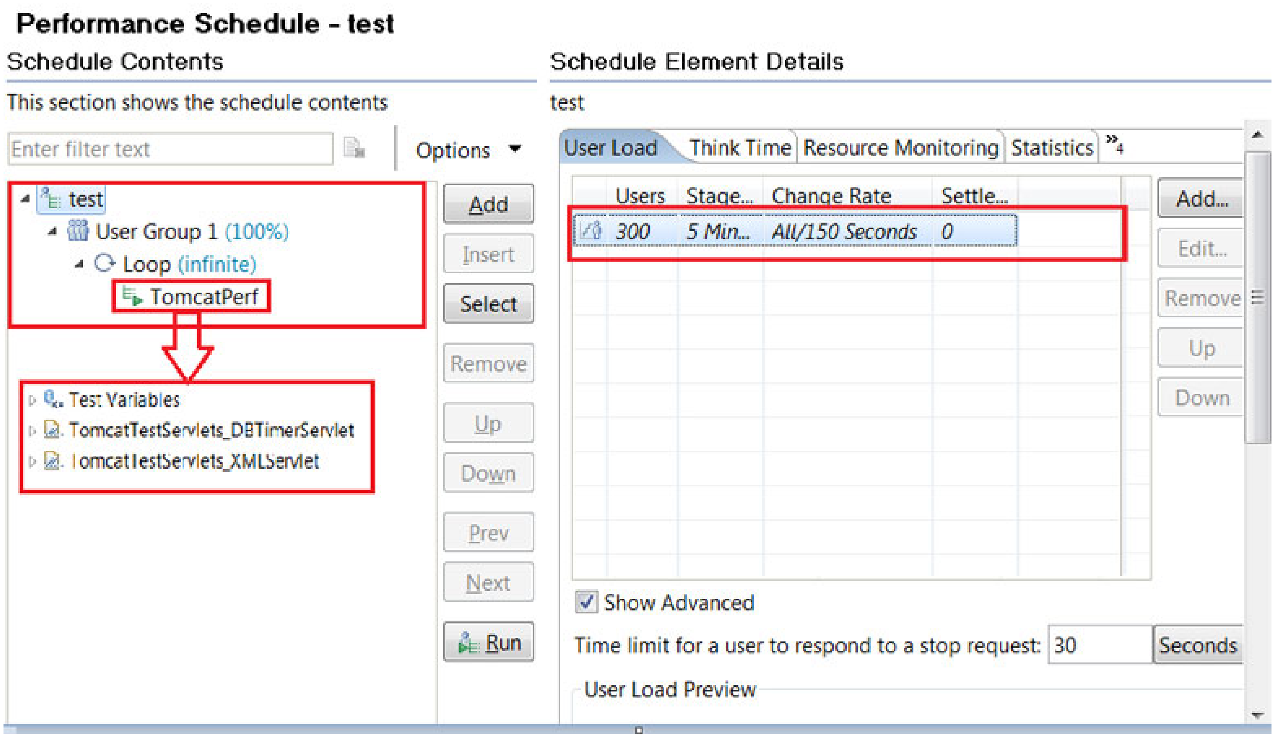
Path: http://192.168.0.101:5555/TomcatTestServlets/DBTimerServlet

☐ Redirect Automatically [✓ Follow Redirects] [✓ Use KeepAlive] ☐ Use multipart/form-data for POST ☐ Browser-compatible headers

After the test ends, you can find the statistics data in the Summary Report, Response Time Graph, and Graph Results sections.

IBM RPT works very well in loading and testing the functional behavior and for measuring performance. With RPT, we can create a performance schedule that simulates a 300-user scenario which runs for 5 minutes. We are not going to cover setting up or deploying RPT; that will be a discussion for another article. We are going to show the power of the tool and see how it can be used to better understand the application's server performance.

Figure 3. Create performance schedule in RPT



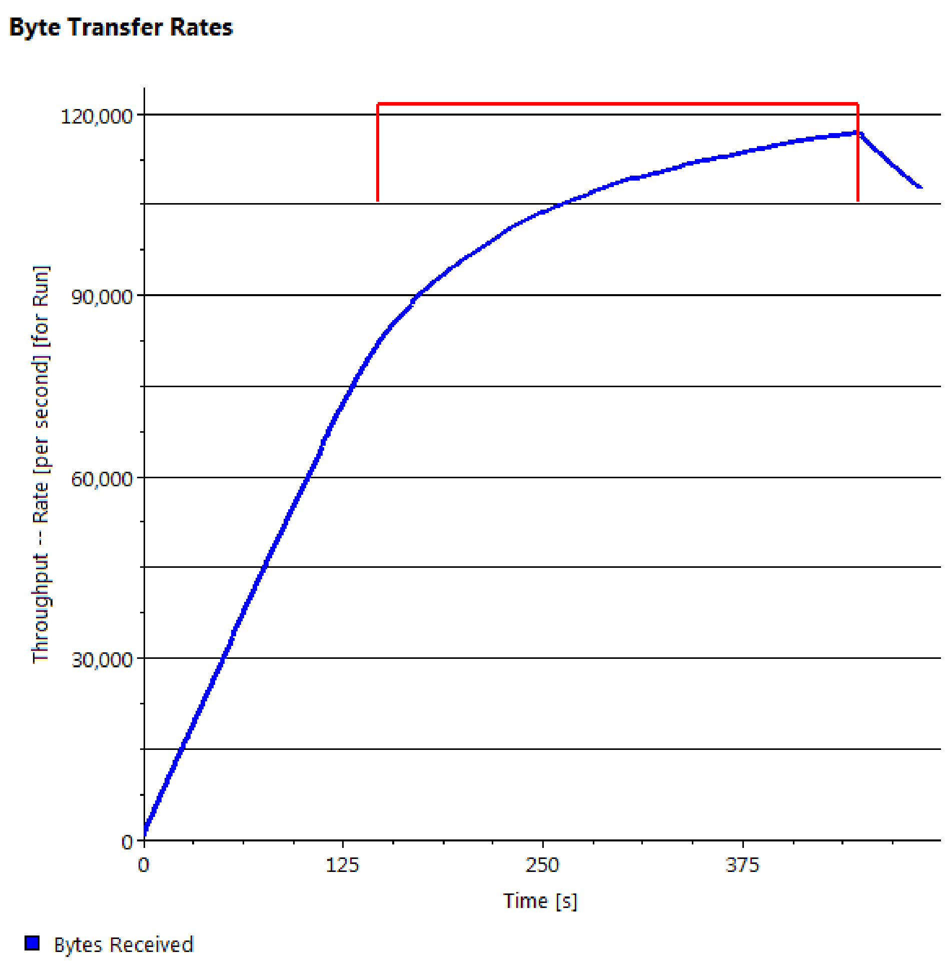
Users	Stage...	Change Rate	Settle...
300	5 Min...	All/150 Seconds	0

Time limit for a user to respond to a stop request: 30 Seconds

To begin with, leave all of the Tomcat and HTTP Server for i related configurations to be set at the default values, and then run the test schedule. This gives us a baseline that we can use to measure any improvements against. Figure 4 and Figure 5 show the throughput and response time baselines.

Throughput statistics:

Figure 4. Throughput statistics before tuning from RPT



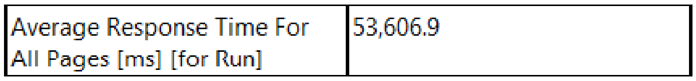
In Figure 4:

- The horizontal x-axis represents the time this workload baseline ran.
- The vertical y-axis represents the real-time throughput, the data retrieved from the server.
- The blue line represents the throughput. At first, all 300 users are being activated. The warm up phase to start all threads covers the first 150 seconds which causes the throughput to increase quickly. The next period is marked by a red fold line and the number of users is steady during this period. This causes the throughput to level off at this period. At the end, all the users are getting inactivated sharply and the throughput decreases for a short period of time.

Response time statistics:

Note: RPT starts to count the response time statistic from the beginning of the test case rather than the time when all the users are activated.

Figure 5. Response time statistics before tuning from RPT



Average Response Time For All Pages [ms] [for Run]	53,606.9

Now, you had a basic overview of the application and the baseline statistics. The following sections describe some general tuning techniques to improve the overall performance of your IBM i web application running on Tomcat.

Running Tomcat in its own subsystem

A subsystem is a single, predefined operating environment through which the system coordinates the work flow and resource used. On IBM i, this is a mechanism that allows you to separate and control workloads. By default, the Tomcat job is running in the QINTER subsystem, which runs all interactive jobs. This works well when there are fewer users and jobs. However, as the number of users and the amount of work on QINTER increase, the QINTER subsystem is sometimes insufficient for this set of work. In addition, leaving your application server in the QINTER subsystem allows for the potential for other interactive workloads (green screen for example) to potentially affect the performance of your web application because both the interactive and application servers are competing for the same resource. As a result, we would like to run the Tomcat job in its own subsystem. Running in a separate subsystem can have the following advantages:

- Improves manageability of the work on the system due to better intellectual control over identifying what work is running on what subsystems.
- Provides the ability to disallow sets of users to access the system at certain periods of time.
- Improves scalability and availability. By having a single subsystem do work for fewer users, the subsystem is less busy and can be more responsive to the work requests it handles.
- Improves error tolerance which is particularly important for interactive jobs such as Tomcat job.
- Improves subsystem startup time.
- Provides better control of the memory required by the application server.
- Uses workload groups to control the amount of processor available to the application server.
- Reduces the potential for other workloads to affect the performance of your application server

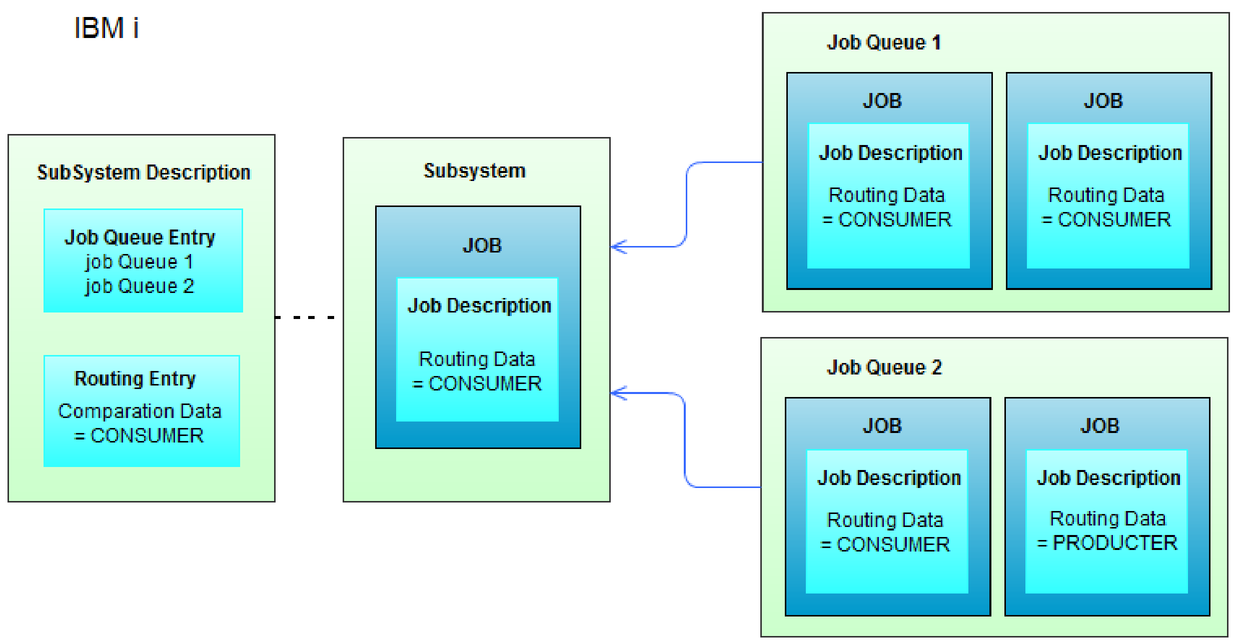
Before starting, let's highlight some key terms that are frequently mentioned:

Table 1. Subsystem related concepts

Term	Description
Subsystem	Defines system resources to run jobs. For example, memory pool and maximum job count.
Subsystem description	Object that contains the attributes of a subsystem. It's used to create and define subsystem.
Job queue entry	Belongs to a subsystem description. Tells the subsystem to pick up jobs in the job queue to run
Job queue	Jobs are submitted into the job queue and then run in the subsystem.

Routing entry	Belongs to the subsystem description. It tells the subsystem jobs in job queue with 'title' (routing data) described here should only be picked up.
Job description	A set of characteristics that are used by the system to define the job. Attributes specified in the job description instruct the system to run the job the way you want.

Figure 6. Overview of the subsystem and job routing on IBM i



After getting familiar with the basic concepts, let's configure your Tomcat application sever to run in its own subsystem. Refer to Figure 6 for the architecture.

Step 1 – Create a new library

Create a new library named TOMCAT7 for storing all IBM i native objects, for example, job queue entry, routing entry, job description and so on.

```
CRTLIB TOMCAT7
```

Step 2 – Create a job queue

This is where the job is submitted before it starts running. Certainly it can be submitted into QBATCH or some other queue, but again, now you are competing with the other work in that queue. We are showing a customized way here that allows maximum control. Create job queue entry named TCJOBQ in the library you created above.

```
CRTJOBQ JOBQ(TOMCAT7/TCJOBQ)
```

Step 3 – Create a job description

This defines how the job should be run. Customize it according to your specific requirements as needed. Create a job description named TCJOB in the above library and associate it with the job queue you just created.

```
CRTJOBD JOBD(TOMCAT7/TCJOB) JOBQ(TOMCAT7/TCJOBQ)
```

Step 4 – Create a class

The class defines the processing attributes for the jobs. The class used by a job is specified in the subsystem description routing entry used when starting the job. We simply need a class here. Create a default one named TCSBS in the library we created above.

```
CRTCLS CLS(TOMCAT7/TCSBS)
```

Step 5 – Create a subsystem description

Here's where the Tomcat server jobs actually run. Create the subsystem with the same name as the class (TCSBS) defined in previous section. The important component that needs to be defined for the subsystem is the memory pool that all the jobs running in this subsystem will use. There are two kinds of memory pools that can be used, base memory pool and private memory pool. The *base memory pool* contains all unassigned main storage on the system that is shared by many subsystems. However, the *private memory pools* (also known as user-defined memory pools) contain a specific amount of main storage that can be used by only a single subsystem. Using a private memory pool means that you can manage the memory for this subsystem without the interference of other subsystems or jobs on the system. Creating a private memory pool is recommended for any workload that you want to make sure is not affected by other work on the same system. If you want to use the base memory pool, run the following CL command:

```
CRTSBSD SBSD(TOMCAT7/TCSBS) POOLS((1 *BASE))
```

If you want to create you own private pool especially for Tomcat, issue the following CL command.

```
CRTSBSD SBSD(TOMCAT7/TCSBS) POOLS((1 1024 500 *MB))
```

Note: In the command, *1024* specifies the size of the new storage pool, *500* specifies the maximum number of threads that can run at the same time in the pool. You can configure your private memory pool according to your requirements. This article creates a private memory pool and sets its pool-size/thread-number to 1024 MB/500. Memory and active threads is a component of application performance that needs to be watched closely. If your application start is memory constrained or is running out of threads, your application performance can be adversely affected. With any Java-based application, memory and the number of available threads is one of the primary factors in causing performance constraints.

Step 6 – Add a job queue entry

The following command tells the subsystem which job queue it should search for jobs to run. Here, we let TCSBS search TCJOBQ.

```
ADDJOBQE SBSD(TOMCAT7/TCSBS) JOBQ(TOMCAT7/TCJOBQ)
```

Step 7 – Add a routing entry

Add the routing entry to the subsystem to define the criteria of job selection and parameters used to start a job.

```
ADDRTGE SBSD(TOMCAT7/TCSBS) SEQNBR(10) CMPVAL(*ANY) PGM(QSYS/QCMD)
```

Step 8 – Start the Tomcat server

All objects have been set up; let's begin testing by starting your Tomcat server in your newly created subsystem.

1. Start the subsystem you created in step 5.
   ```
   STRSBS SBSD(TOMCAT7/TCSBS)
   ```
2. Submit the job into the customized subsystem to start the Tomcat application server. In our case, the script to start Tomcat is located in /home/download/apachetomcat7.0.28/bin/startup.sh
   ```
   SBMJOB JOBQ(TOMCAT7/TCJOBQ) JOBD(TOMCAT7/TCJOB) CMD(QSH CMD(
         '/home/download/apache-tomcat-7.0.28/bin/startup.sh’))
   ```

Step 9 – Create a customized CL command to start the Tomcat server

By default, we need to run two complicated commands to start the subsystem and submit the jobs to start the Tomcat application server. This is inconvenient. Let's create a C program and tie it to a CL command called STRTMC to make it easier.

1. First, create a source code file [strtomcat7.c (refer to the Downloadable resources section), located in /home/downloads for this example] to start the subsystem and run the tomcat startup script. Below is the code for this program.
   ```
   #include <stdio.h>
   #include <unistd.h>
   #include <stdlib.h>
   #include <sys/stat.h>

   int main(int argc, char ** argv)
   {
           char cmdbuf[4096];
           //make command string, pass the parameter to script.
           sprintf(cmdbuf, "SBMJOB JOBQ(TOMCAT7/TCJOBQ) \
           JOBD(TOMCAT7/TCJOB)\
           CMD(QSH   CMD('/home/download/apache-tomcat-7.0.28/bin/startup.sh'))");

           // we do not expect the output screen (default console) with
           // a "press any key to continue". This program should end immediately.
           umask(0);

           // ensure subsystem is up
           system("STRSBS SBSD(TOMCAT7/TCSBS)");

           // submit the job
           system(cmdbuf);
   ```

```
        return 0;
}
```

2. Create the C program.

   ```
   CRTBNDC PGM(TOMCAT7/STRTMC) SRCSTMF('/home/download/strtomcat7.c')
   TEXT('Tomcat start script invoker')
   ```

   Here's an example of the content:

   ```
   *************** Beginning of data*************************************
   0001.00 CMD    PROMPT('START TOMCAT7')
   ****************** End of data****************************************
   ```

3. Create our customized CL command.

   ```
   CRTCMD CMD(TOMCAT7/STRTMC) PGM(TOMCAT7/STRTMC)
   SRCFILE(TOMCAT7/CMDSRC) SRCMBR(STRTMC)
   ```

   This command is named STRTMC and is located in the TOMCAT7 library. It actually calls the program STRTMC in TOMCAT7.

4. Run the STRTMC command to start the Tomcat application server.

   ```
   TOMCAT7/STRTMC
   ```

The Tomcat jobs are now running in the customized subsystem TCSBS

The steps illustrate how to create a customized CL command to start and run the Tomcat application server in its own subsystem. You can refer to these steps and make modifications to create another CL command, ENDTMC, to make it convenient to stop the Tomcat application server (and you can find the source code, endtomcat7.c, in the Downloadable resources section of this article).

Tuning the JVM

The Tomcat application server is based on Java and runs on the JVM. This means that the JVM may have some impact on Tomcat's overall performance. The following section introduces recommendations for improving the web application's performance by optimizing the JVM.

Note: This example in this article is based on IBM i 7.1. The JVM on IBM i is IBM Technology for Java. Here we do some basic tuning for IBM Technology for Java.

For example, we can edit the configuration file /home/download/apachetomcat7.0.28/catalina.sh and add the following line in this file to configure some of the JVM options.

```
JAVA_OPTS="$JAVA_OPTS -Xquickstart -Xms2048M -Xmx2048M
```

Below is the description of the attributes that have just been set.

- *-Xquickstart:* Xquickstart is used to cause the just-in-time (JIT) compiler to run with a subset of optimizations which makes short-running applications perform better. In this article, this directive is enabled (the default value is disabled).
- *-Xms2048M –Xmx2048M:* Xms and Xmx are used to define the heap size used by the JVM.

Xms defines the initial size of the heap and Xmx defines the maximum size of the heap.

If memory is adequate, you can set the two attributes to the same value to meet the maximum memory requirement of the application to avoid re-allocation of memory after garbage collection. In this article, the two attributes are set to 2048 MB because the sample in this article demands a large amount of memory during peak loads. However, if memory is limited, you can set Xms to meet the *average* memory requirement of the application and Xmx to meet the *maximum* memory requirement of the application. For example, you can set the Xms to 512 MB and Xmx to 2048 MB.

Tuning Tomcat

The Tomcat server has many tuning parameters that can be adjusted to significantly improve the web application's performance (or if you are not careful, you might hurt the application's performance!). This section introduces some settings defined in the Tomcat's configuration files.

Table 2. Key terms

Term	Description
Connector	Represents the interface between external clients sending requests to and receiving responses from a particular service.
Containers	Represent components whose function is to process incoming requests, and create the corresponding responses.
Host	Is a container that handles all requests for a particular virtual host
Context	Is a container that handles all requests for a specific web application

This section explains the tuning techniques for these components. We are going to open a number of different files and update some of the important attributes found in these configuration files.

- **Connector** (/home/download/apachetomcat7.0.28/conf/server.xml)

```
<Connector port="8009" Protocol="AJP/1.3"
 connectionTimeout="20000" redirectPort="8443" maxThreads="300" maxSpareThreads="100"
 minSpareThreads="40" maxKeepAliveRequests="200" socketBuffer="12000"
 acceptCount="300" enableLookups="false" compression="on"
 compressableMimeType="text/html,text/xml,text/javascript,text/css,text/plain"
/>
```

  - `maxThreads=<val>:` This attribute specifies the maximum number of request processing threads to be created by this connector. This determines the maximum number of simultaneous requests that can be handled. The default value for this attribute is 200. There is no optimum value for this attribute because too many threads might cause other costs due to processors' switch operation. For this example, the attribute is set to 300. This is an attribute that may require trail and error to determine the optimum value.
  - `maxSpareThreads=<val>:` This attribute specifies the maximum number of unused request processing threads that are allowed to exist until the thread pool starts stopping the unnecessary threads. The default value for this attribute is 50. It is recommended to assign this attribute a relatively high value to cope with emergences. For this example, the attribute is set to 100.
  - `minSpareThreads=<val>:` This attribute specifies the number of request processing threads that are created when this connector is first started. The default value of this

attribute is 25.It is recommended to assign this attribute a relatively high value. For this example, the attribute is set to 40.

- `maxKeepAliveRequests=<val>:` This attribute specifies the maximum number of HTTP requests that can be pipelined until the connection is closed by the server. The default value of this attribute is 100. It is recommended to assign this attribute a relatively high value to enable more HTTP requests to be pipelined. For this example, the attribute is set to 200.
- `socketBuffer=<val>:` This attribute specifies the size of the buffer to be provided for socket output buffering. The default value of this attribute is 9000 bytes. It is recommended to assign this attribute a relatively high value to generate a good buffer. For this example, the attribute is set to 12000 bytes to cope with high-level peak load data.
- `acceptCount=<val>:` This attribute specifies the maximum queue length for incoming connection requests when all possible request processing threads are in use. The default value of this attribute is 100.It is recommended to assign this attribute a relatively high value. For this example, the attribute is set to 300.
- `compression=<val>:` This attribute specifies whether or not to compress the data to be transferred. This attribute and the `compressableMimeTypes` attribute are used together. Although the two attributes can reduce the data transferred, we can not promise that they can accelerate the data transferring speed because the compression action costs time too. As a result, you have to do some testing with different values to see whether these two attributes do save time. If not specified, this attribute is set to off. For this example, the compression is set to on.
- `compressableMimeTypes=<val>:` This attribute specifies what to compress. The default value is text/html,text/xml,text/plain. For this example, the attribute is set to "text/html,text/xml,text/javascript,text/css,text/plain".

- **Host**`(/home/download/apachetomcat7.0.28/conf/server.xml)`

```
<Host name="localhost" appBase="webapps"
    unpackWARs="true" autoDeploy="false">
```

  - `autoDeploy=<val>:` This attribute indicates if Tomcat should check periodically for new or updated web applications while Tomcat is running. Updated web applications or context XML descriptors trigger a reload of the web application. The default value of this attribute is `true`. For a development environment, you can set this attribute to on. However, it is suggested to set this attribute to `false` in a production environment.

- **Context**`(/home/download/apachetomcat7.0.28/conf/context.xml)`

```
<Context swallowOutput="true" reloadable="false">
```

  - `swallowOutput:` This attribute directs the output to System.out and System.err by causing the web application to be redirected to the web application logger. We would like to set this attribute to `true` to make it easier to troubleshoot web application problems.
  - `reloadable=<val>:` This attribute controls the Java™ classes hot deployment for Tomcat. It makes Tomcat monitor classes in /WEB-INF/classes/ and /WEB-INF/lib for changes, and automatically reloads the web application if a change is detected. The default value for this attribute is `false`. For development environment, you can set this value to `true` to

make it convenient to develop. However, it is suggested to leave this attribute as `false` in a production environment because it can slow the runtime workload significantly.

Tuning the HTTP server

The HTTP server is a general-purpose web server. It is designed to provide a balance of flexibility, portability, security and performance. HTTP Server for i contains many additional optimizations to increase throughput and scalability for our IBM i environment. Most of these improvements are enabled by default. This section describes the options that can be configured to tune the HTTP Server for i to improve the web application's performance.

Refer to the HTTP server configuration for our example application. You can find the configuration file is at '/www/<http instance name>'. The attributes in bold are the attributes we are going to review.

```
LoadModule jk_module /QSYS.LIB/QHTTPSVR.LIB/MOD_JK.SRVPGM
JkWorkersFile /www/httpserver/conf/workers.properties
JkLogFile /www/httpserver/logs/jk.log
JkLogLevel warn
JKMount /manager/* worker1
JKMount /TomcatTestServlets/* worker1
Listen *:5555
DocumentRoot /www/httpserver/htdocs

ThreadsPerChild 120
AcceptThreads 12
LogLevel warn
HostNameLookups off
CacheLocalFile /home/download/apachetomcat7.0.28/webapps/TomcatTestServlets/data/*.xml
CacheLocalFile /home/download/apachetomcat7.0.28/webapps/TomcatTestServlets/data/*.xsl
DynamicCache on
LiveLocalCache off
CacheLocalSizeLimit 51200
CacheLocalFileSizeLimit 90000
```

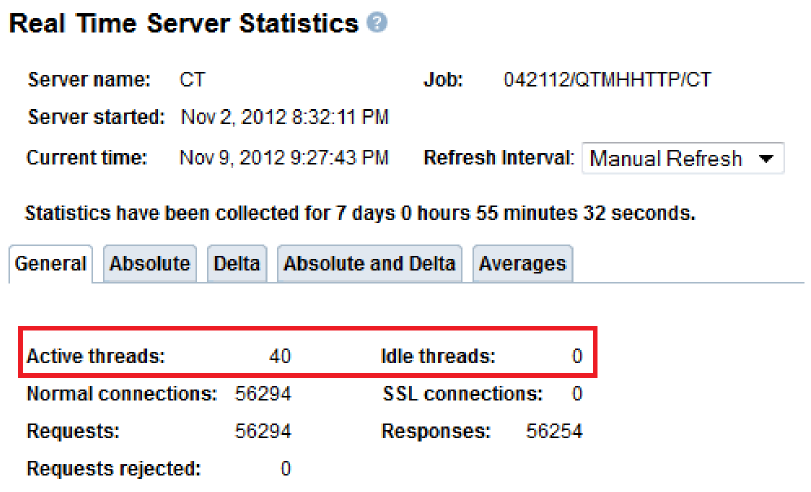
`ThreadsPerChild:` This directive is used to specify the maximum number of threads per server child process. If you do not specify a value for the directive, it inherits the global HTTP server setting (default value is 40 and can be changed using the `CHGHTTPA` command). Users should set the value according to the real requirement. The value can impact performance a lot. For example, a smaller value results in HTTP server responding slowly and a higher value results in wasting resources.

A child process creates threads at startup and never creates more. This number should be high enough to handle the number of simultaneous connections anticipated during peak HTTP server load. Care should be taken to consider system memory and processing capacity when increasing this directive's value. Setting this attribute too high can adversely affect the HTTP server and system performance by consuming a high amount of system resources. If the maximum number of server threads is reached during peak load, additional connection requests are placed on the sockets connection request backlog until a server thread becomes available to service the connection request. With the server's use of asynchronous I/O, each server thread can only handle connections that have received a request. The thread can handle other connections while waiting for new requests on an existing connection.

`AcceptThreads:` This directive specifies the maximum number of accept threads per server child process. If you do not specify a value for this directive, the server uses a limit of four accept threads per process. The accept threads are used to accept new connections from the client, and then forward those requests to the HTTP server worker threads to handle. This directive's value might need to be changed to reflect the number of concurrent connections that are being accepted. If there are a large number of connections to the web server starting approximately at the same time, this number might need to be adjusted to a higher value. These accept threads are created at startup time, and the process never creates more. (The default value is 4, and the maximum number is 20).

You can use the **Real Time Server Statistics** link in the Web Administration GUI to see how many idle threads are available before tuning the HTTP server. If there are no idle threads available for a long continuous time after all the 300 users are activated, indicating that the server is very busy, you can consider changing the `ThreadsPerChild` directive to a higher value.

Figure 7. Real-time server statistics before tuning



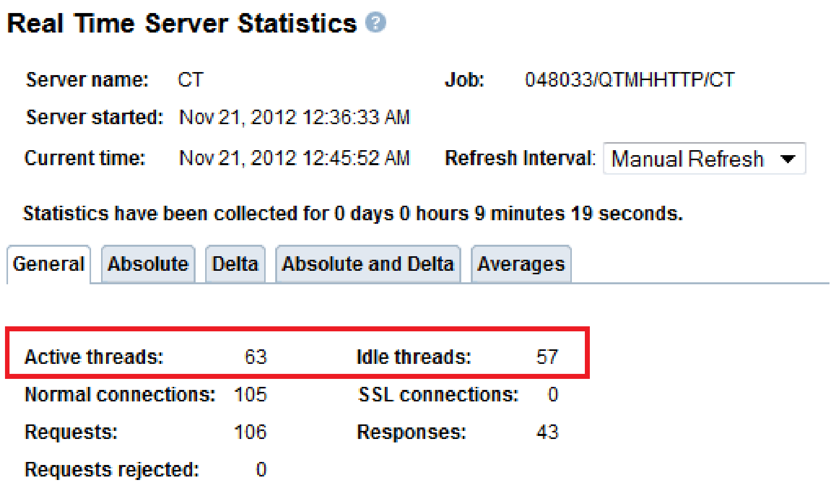
Server name: CT
Job: 042112/QTMHHTTP/CT
Server started: Nov 2, 2012 8:32:11 PM
Current time: Nov 9, 2012 9:27:43 PM
Refresh Interval: Manual Refresh

Statistics have been collected for 7 days 0 hours 55 minutes 32 seconds.

Active threads:	40	Idle threads:	0
Normal connections:	56294	SSL connections:	0
Requests:	56294	Responses:	56254
Requests rejected:	0		

This indicates that the HTTP server is keeping busy for very long time and all new client requests need to wait until a thread becomes available. The recommendation for this condition is to increase the value of the `ThreadsPerChild` directive to a more reasonable number. For this example, the attribute is set to 120 to make sure that there are sufficient idle threads when the application runs up to its peak load. Users can set this directive according to their application's actual requirement. Figure 8 shows the real-time server statistics after tuning this attribute.

Figure 8. Real-time server statistics after tuning


Real Time Server Statistics

Server name: CT   Job: 048033/QTMHHTTP/CT
Server started: Nov 21, 2012 12:36:33 AM
Current time: Nov 21, 2012 12:45:52 AM   Refresh Interval: Manual Refresh

Statistics have been collected for 0 days 0 hours 9 minutes 19 seconds.

Active threads:	63	Idle threads:	57
Normal connections:	105	SSL connections:	0
Requests:	106	Responses:	43
Requests rejected:	0		

`LogLevel:` This directive adjusts the verbosity of the messages recorded in the error logs. For development environments, you can set this directive to a relatively low value, for example, debug or info to generate detailed log messages for finding errors and debugging issues. However, for production environments, you should set this directive to a high value to reduce the detail specified in the log file. A low log level can influence the web application's performance. For this example, the directive is set to its default value (warn) as if it was a production environment.

`HostNameLookups=<val>:` The directive enables DNS lookups and causes the HTTP server to do a reverse lookup to convert an IP address into a host name and domain. The default value for this directive is `off` to save on the network traffic for those sites that do not truly need the reverse lookup. However, this directive can be set to `on` if DNS lookups is really needed (for example, users who need to record the domain names in the access logs). For this example, this directive is set to its default value `off` because DNS lookups can take a considerable amount of extra time and resources.

If you have static files such as HTML, JPEG, JS, and CSS that are visited frequently and do not change a lot, you can allocate caches files for these files by setting the following directives below. If not, you can just omit these directives. Putting these types of static objects in a cache can save a great deal of time doing file look up.

`CacheLocalFile:` This directive is used to specify the name of files that you want to load into the server's memory each time you start the server. You can have multiple occurrences of this directive in the configuration file allowing many files to be specified. By keeping the most frequently requested files loaded in the server's memory, you can improve your server's response time for those files. For example, if you load your server's welcome page into memory at startup, the server can handle requests for that page much faster than if it had to read the file from the file system every time. In the sample application, XML and XSL files are cached.

`DynamicCache:` This directive is used to specify whether you want the server to dynamically cache frequently accessed files. Setting the dynamic cache directive to `on` instructs the server to cache the most frequently accessed files, which results in better performance and system throughput.

`LiveLocalCache:` This directive is used to specify whether the cache is updated when a cached file is modified. When it is set to `on`, before responding to a request for a file that is stored in memory, the server checks to see if the file has changed since the server was started. If the file has changed, the server responds to this request with the updated file and then deletes the older file version from memory. Set this directive to the default value of `on` if you want users, requesting a cached file, to receive the file with the latest updates. Setting this directive to `off` is the optimum setting for performance.

`CacheLocalSizeLimit:` This directive is used to specify the maximum amount of memory, in Kilobytes, that you want to allow for file caching. You must specify the files that you want cached with the *CacheLocalFile* directive or by setting *DynamicCache* to on. The storage is allocated as files are cached. For this example, the directive is set to 4000 instead of the default value 2000 to cope with heavy load. Users can set this directive according their requirement.

`CacheLocalFileSizeLimit:` This directive is used to specify, in bytes, the largest file that will be placed in the local memory cache. A file larger than the value specified for `CacheLocalFileSizeLimit` is not placed in the cache. This prevents the cache from being filled by only a small number of very large files. For this example, the directive is set to 120000 instead of the default value, 90000.

Fast Response Cache Accelerator (FRCA) is another useful caching technique. It runs completely under IBM i machine interface (MI) that allows FRCA to perform efficiently as a System Licensed Internal Code task which avoids the costly work of switching to a user-level server thread such as the case for the HTTP Server for i. FRCA can serve web content either in the form of a local cache for static content or in the form of a reverse proxy cache for dynamic content. FRCA performs much better than the traditional cache techniques when serving web contents. However, FRCA does not support Secure Sockets Layer (SSL) and Transport Layer Security (TLS). Additionally, FRCA can not protect authenticated files from unauthenticated user access. So, if your web application's content does not need to be secured or accessed through a specific validation, FRCA is the fastest choice. You can also consider using a blend of approaches.

All the above tuning techniques are just the tip of the iceberg on performance tuning of HTTP Server for i. You can refer to chapter 10, *Getting the best performance from HTTP Server* in the HTTP Server for i Redbooks and other materials if you want to investigate further.

Tuning web applications

Now that we have done as much as we can to customize Tomcat's relevant configurations to improve the web application's performance, it is time to look at the improvements of the web application itself. This process is more complex, but the performance gains can be exponentially greater.

The best time to think about optimizing the web application's performance with Tomcat is during the development phase. So, consider all the design choices carefully. You can also refer to the following general tips.

- **JSP pre-compilation**: Pre-compile JSP files before deploying the web applications. This means that a JSP file does not need to be compiled when running it the first time. It can also protect the source code of the JSP files. The alternative is to allow the JSP files to be compiled dynamically on first touch.
- **Cache dynamic page output**: If each request for a page generates the same output, consider temporarily caching the generated output.
- **Tactics selection**: Consider factors such as the suitability of protocol for the project. For example, if you are loading large amounts of data and need high performance, XML might not be the appropriate choice.
- **Database connection pooling**: DB connection pool can significantly influence the web application's flexibility and robustness.
- **Database object caching**: Using middleware to persist and cache objects from database can significantly improve performance.

With all these techniques, you are well on your way to great Tomcat performance on application level.

Tuning effect

After completing all these tuning techniques, let's verify whether the web application's performance is better. Generate statistics with exactly the same testing case using RPT. Figure 9 and Figure 10 show the statistics of the throughput and response time.

Figure 9. Throughput statistics after tuning from RPT

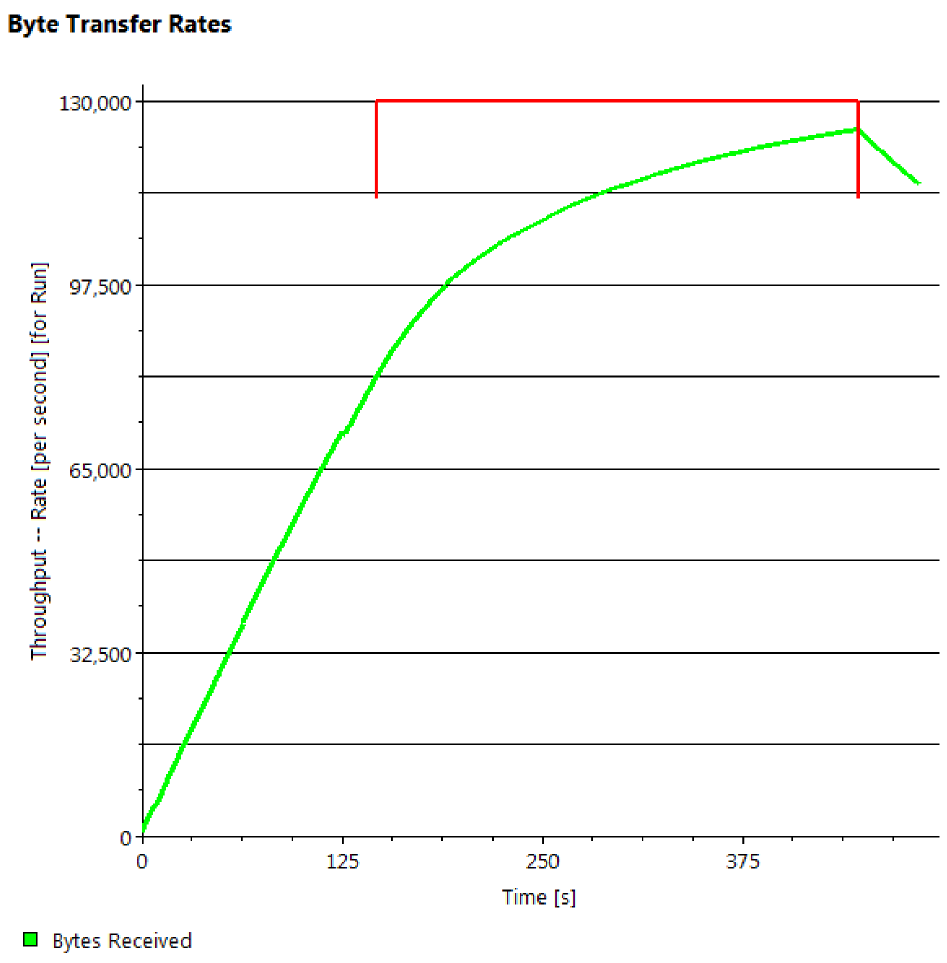
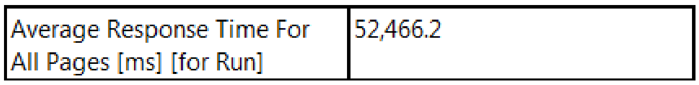


Figure 10. Response time statistics after tuning from RPT

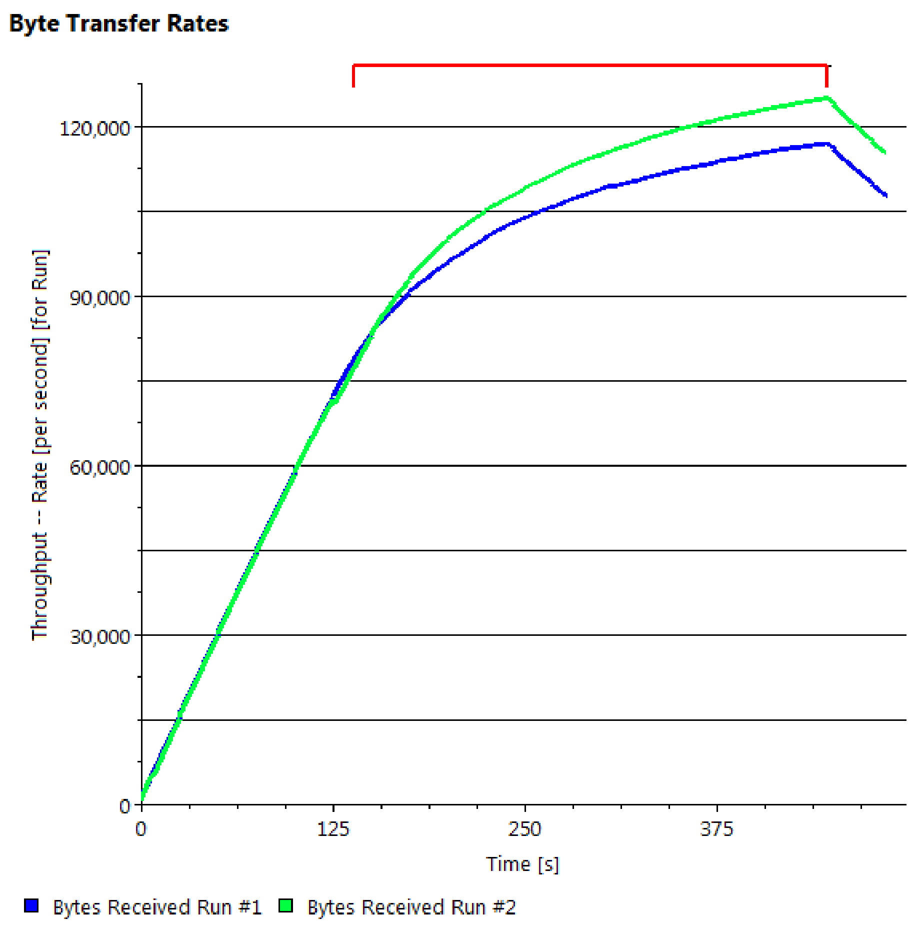


Average Response Time For All Pages [ms] [for Run]	52,466.2

Now compare the statistics before and after Tomcat tuning using RPT. Figure 11 shows the comparison statistics.

Throughput statistics:

Figure 11. Throughput comparison statistics before and after tuning from RPT

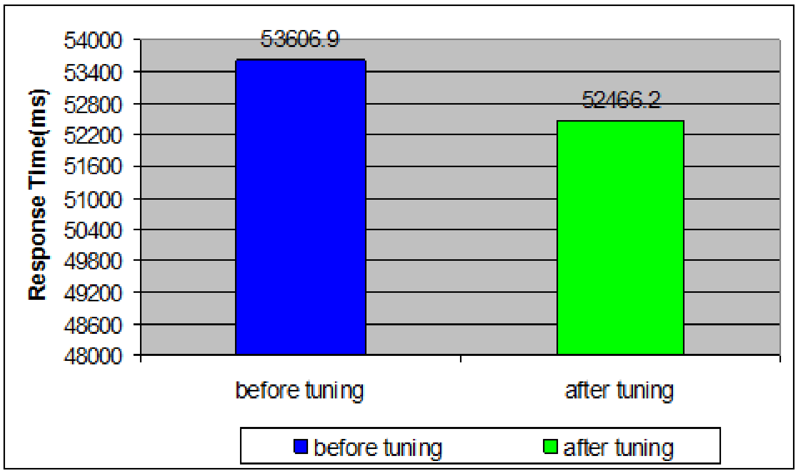


In Figure 11:

- The horizontal ordinate represents the time this baseline cases spend.
- The longitudinal ordinate represents the real-time throughput.
- The red line represents the period that a stable number of users are sending or receiving data.
- The blue curve represents the throughput before tuning.
- The green curve represents the throughput after tuning.

Response time statistics:

Figure 12. Response Time statistics comparison before and after tuning



It can be seen from Figure 12 that at a stable phase (all users are activated), the throughput is increased by about 7% and the response time is deduced by about 2%. It proves that the web application's performance on IBM i is surely improved after all the tuning techniques.

Summary

These performance adjustments are intended to help you better understand the many factors that can impact your web applications. The only way to determine what works best in your environment is to invest in the necessary performance evaluation. Much is done through trial and error methods. To help you know when to stop, be sure you understand what your performance requirements are before you start tuning.

Resources

- Tomcat 6 Tunning
- IBM HTTP Server Performance Tuning
- Apache Performance Tuning
- So You Want High Performance
- Running your IBM i web solution on ASF Tomcat

Downloadable resources

Description	Name	Size
	TomcatTestServlets.zip	7.08 MB
	endtomcat7.c	1 KB
	strtomcat7.c	1 KB