Web Test Tech AEM

This test source code is available on my github repository: https://github.com/leonacilibeanu/WebServer. The WebTechTestAEM project is divided in two packages: the servers package (with the web sever implementation) and the test package (with the tests source code).

Thread Pooled Server

For the web server implementation I chose to use a sample code[1] that I previously found over the Internet.

I implemented a short code sample to start the server. It can be found in the Main class in servers package. The web server runs on the localhost on port 9000. So, you can test it by simply accessing the link: http://localhost:9000/.

After starting the server, it creates a pool of threads that will be able to manage the connections which will be stored in a queue. If the number of connections outreaches the number of threads in the pool of threads, then it will have to wait in the queue until a thread becomes available.

Each connection will be wrapped in a Runnable that verifies the input and implements the keep-alive behaviour. If the input is correct, then it will send back a message, in order to be displayed. The web server wasn't meant to display any fancy design, so all you will see it will be a text message.

The web server accepts connections that are sent only over HTTP, so you will not be able to connect over HTTPS or any other protocol.

Load testing

The objectives, in order to perform a load testing, were: response time, bandwidth, resource utilization.

To measure the response time, I simulated different loads for the sever using MapReduce.

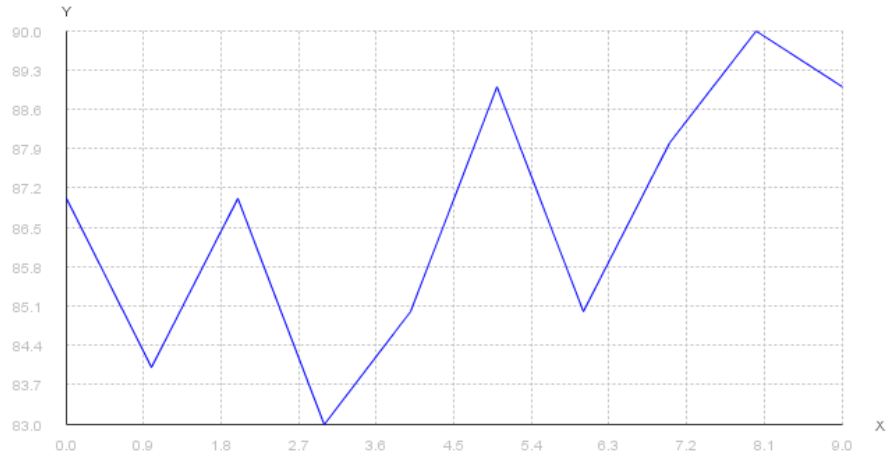
By instantianing the ResponseTimeMaster, it creates a pool of threads and using the map function submits the number of threads given as a parameter to the constructor. It will wait for the threads to finish and returns a list with the reponse times.

The submitted threads are ResponseTimeMap instances. Every instance makes an HTTP connection to the server and waits for the response. The response time is measured and returned in the map function to be added at the result list.

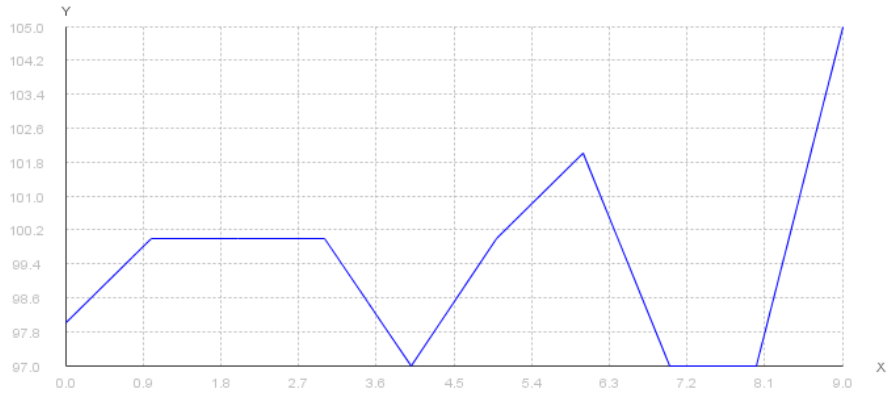
Now, the partial solution from the mapper is sent to the reducer (reduce function in ResponseTimeMaster) which creates an instance of ResponseTimeReduce

for every entry of the list. Each reducer purpose is to write to a file the response times that were previously computed.

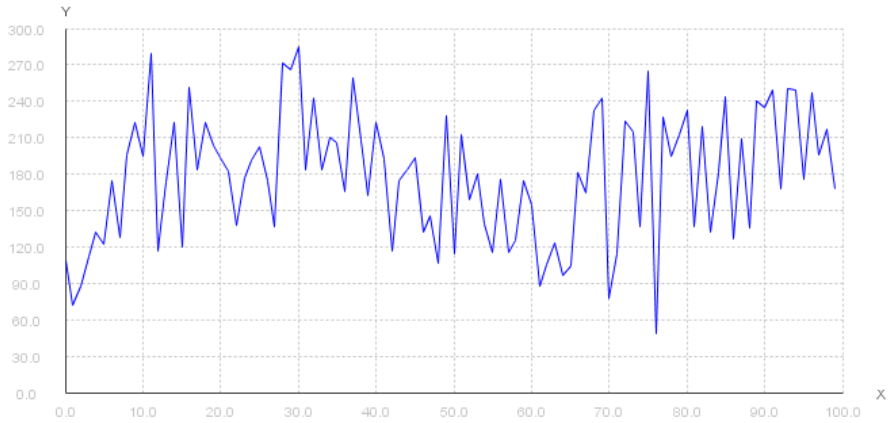
In the WebServerLoad class, the *showResponseTime()* function reads the data from file and creates a plot using this measurements. To create the plots I used a code sample[2] and an additional library: jmathplot.jar.



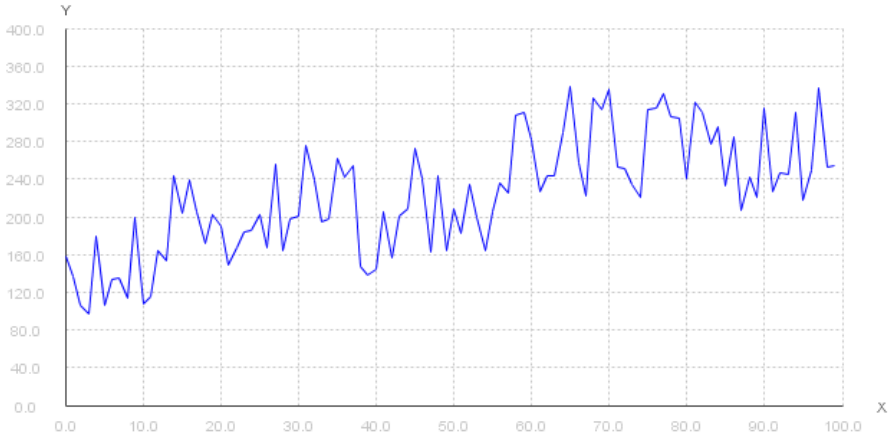
Pool size : 10, Num threads: 10



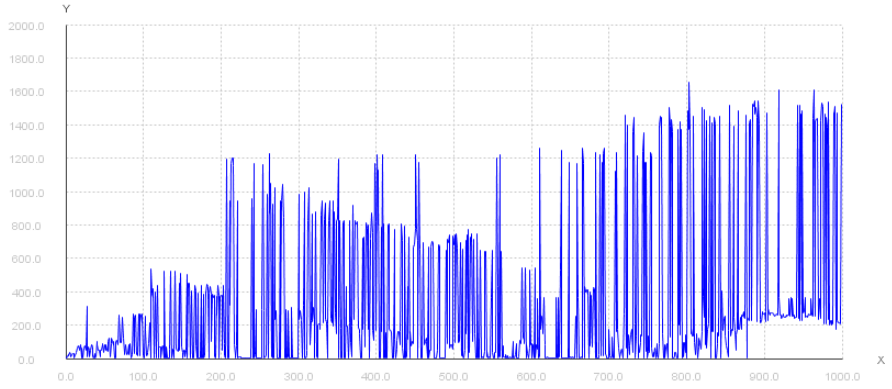
Web Server Pool size : 100, Num threads: 10



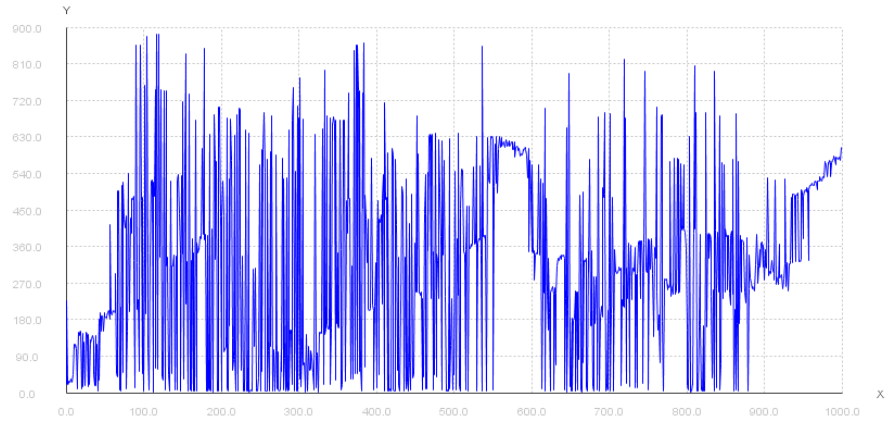
Web Server Pool size : 10, Num threads: 100



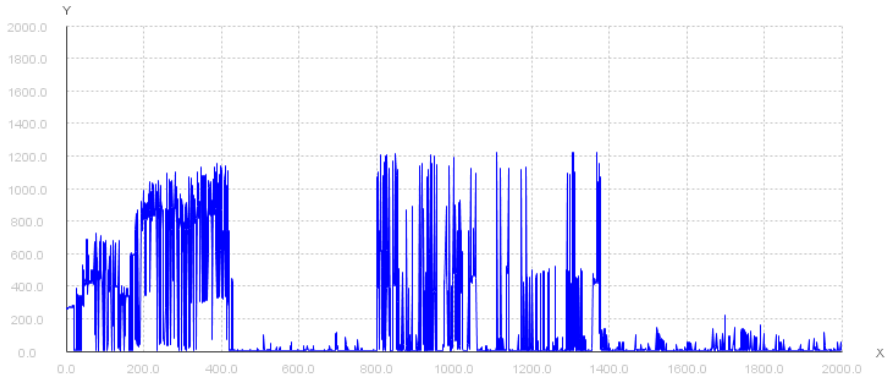
Web Server Pool size : 100, Num threads: 100



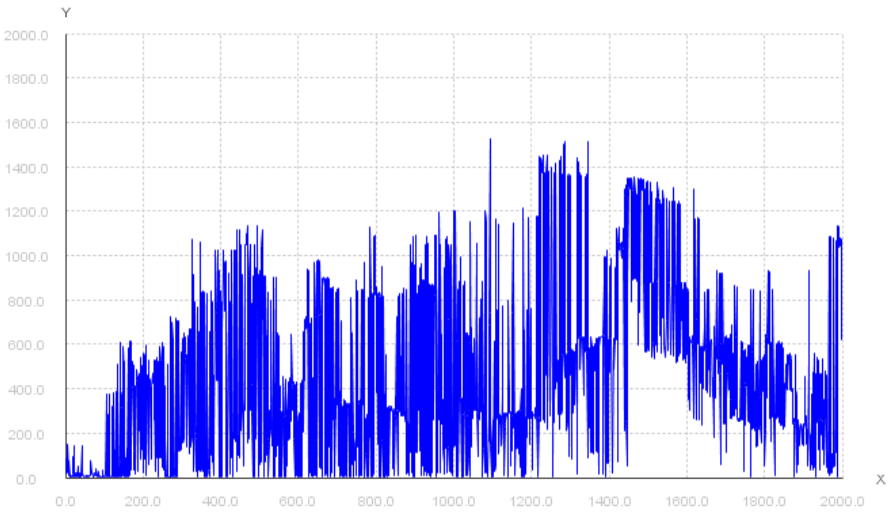
Web Server Pool size : 10, Num threads: 1000



Web Server Pool size : 100, Num threads: 1000



Web Server Pool size : 10, Num threads: 2000



Web Server Pool size : 100, Num threads: 2000

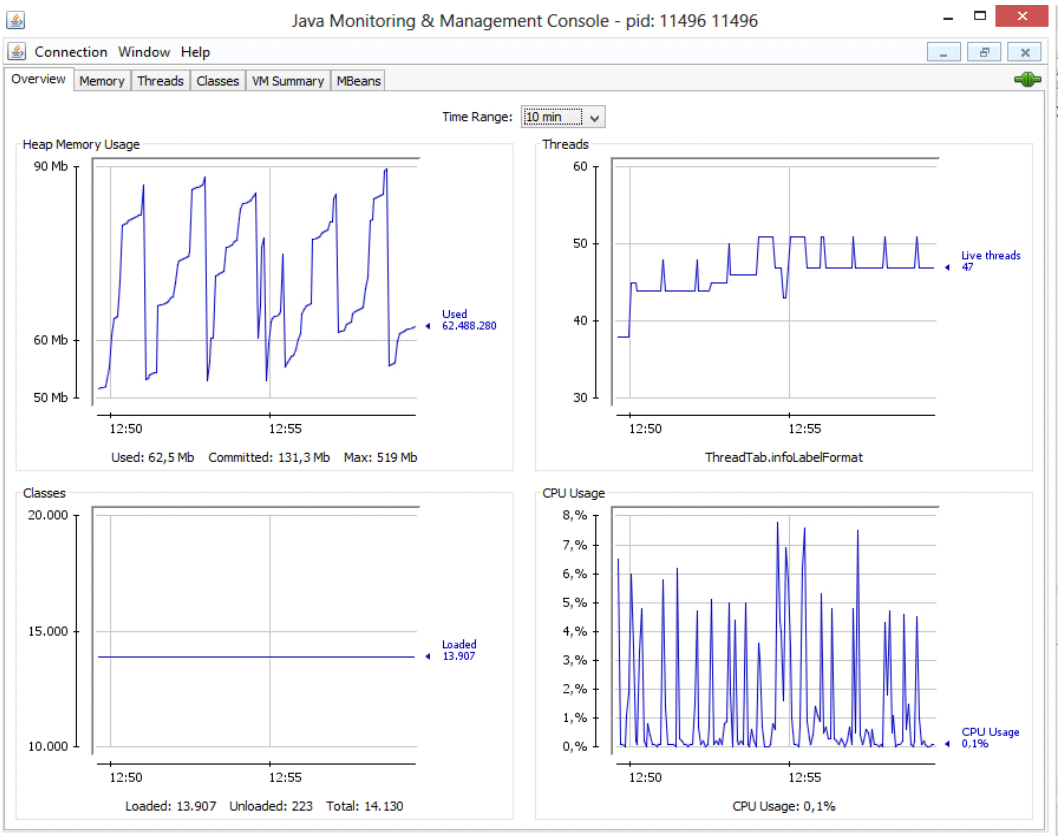
Having the file with the response times and a constant length of the message received from the web server we can now compute the average bandwith. The function *getBandwidth(),* from WebServerLoad class computes this value and writes it to a file.

Here is the average bandwidth for the test cases that I ran above.

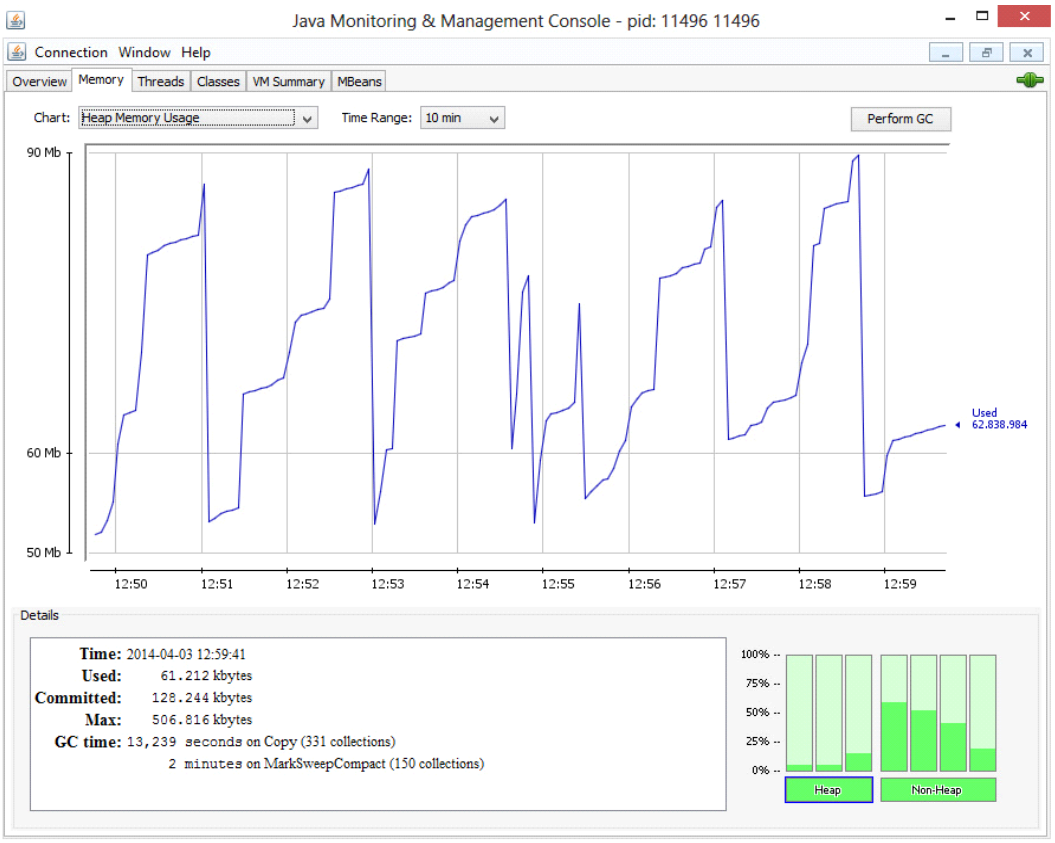
Web Server Pool size / Num threads	10	100	1000	2000
10	702 kb/s	411 kb/s	3569 kb/s	11716 kb/s
100	652 kb/s	436 kb/s	1683 kb/s	3272 kb/s

To see the resource utilization I used JConsole. I ran tests in two cases. First time, the web server thread pool size was 10 and the second time was 100. For each configuration, the server loading was: 10, 100, 1000, 2000, 2500 threads. For the first five tests the loading was ascending, whereas for the next five it was descending. So, you can see in the threads plot that the ones in the middle last longer to complete and, also, the CPU usage is incresed.

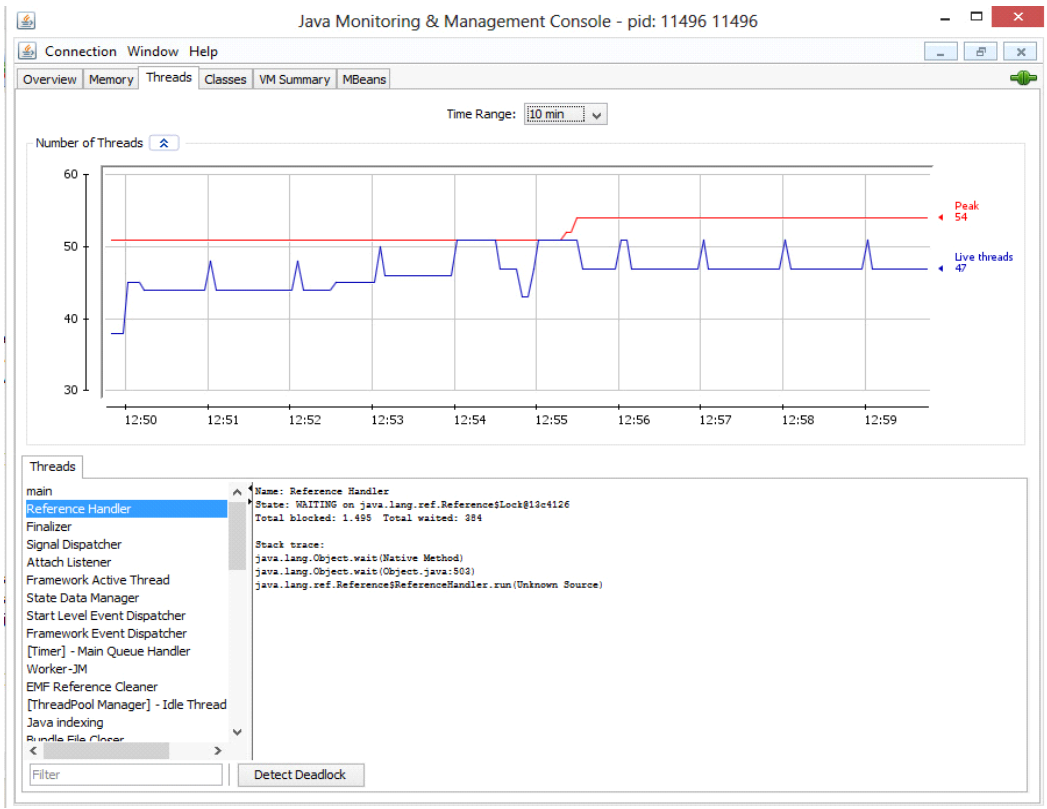
Below, you can see the outputs.



Overview



Heap Memory Usage



Threads

Note: In order to run this load test, you must first start the server by running the Main class from servers package and then, run WebServerLoad Class from test package. You can modify the number of threads or the pool of threads size from Contants Class.

JUnit testing

For unit testing, I implemented four different tests. Three of them check the functionality of the server, whereas the fourth one was meant to test the maximum server load. There are located in Tester class, in test package.

The first test, simply tries to connect and verifies if the response is the expected one.

The second test, verifies the keep-alive behaviour by sending a request with the Connection parameter set to "close" value. It is expected to fail. Which it does.

In the third test, it tries to connect to the server over HTTPS. As the previous one, this one is expected to fail too.

The fourth one uses a part of the MapReduce behaviour that I implemented for load testing. Using the ResponseTimeMaster, creates a pool of threads and with the map function, submits the number of connections that are sent as a parameter to the ResponseTimeMaster. I tried to test it using as many threads as I could, but anything over 2700 threads will fail due to the insufficient memory of the JVM.

In Tester Class, are also implemeted a BeforeClass and an AfterClass which are meant to start and stop the web server.

Sample bug report

Bug Name: Remote host closed connection during handshake
Bug ID: 5000
URL: https://localhost:9000/
Severity: HIGH (High/Medium/Low) or 1
Priority: HIGH (High/Medium/Low) or 1
Assigned to: Leona
Reported By: Leona
Reported On: 04/04/2014
Reason: Defect
Status: New
Environment: Windows 8
Description: Server connection failed while trying to access the URL.
Steps To Reproduce: Run Tester Class.

Bug Name: Unable to create new native thread
Bug ID: 5001
URL: hhtp://localhost:9000/
Severity: MEDIUM (High/Medium/Low) or 1

Priority: MEDIUM (High/Medium/Low) or 1
Assigned to: Leona
Reported By: Leona
Reported On: 04/04/2014
Reason: Defect
Status: New
Environment: Windows 8
Description: In function testLoad(), from Tester Class, couldn't load server with 3000 concurrent connections.
Steps To Reproduce: Set the parameter for ResponseTimeMaster instance to 3000, run Tester Class.

References

[1] http://tutorials.jenkov.com/java-multithreaded-servers/thread-pooled-server.html
[2] http://code.google.com/p/jmathplot/