Testing load on Multi-tier Web Applications

Michael Brevard
Keshav Dasu
Caleb Martinez
Zachary Sells
Martin Ting

Table of Contents

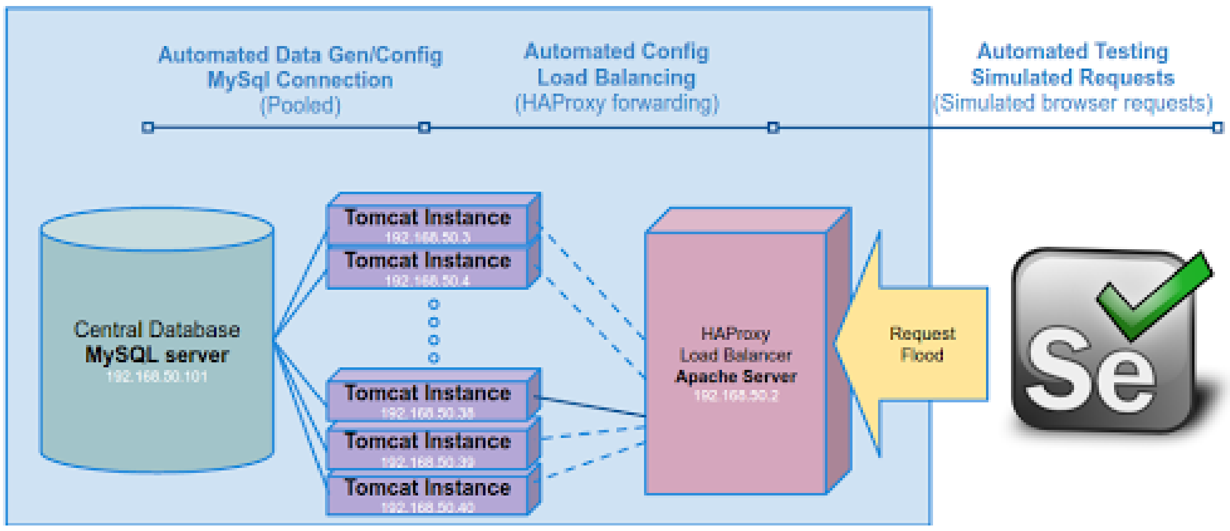
Section 1: Introduction	3 - 5
Project Description	3
System Architecture	3
Programs Used	4 - 5
Section 2: Testing Procedure	6
Description of Tests	6
Section 3: Results	7 - 9
Raw Data (Tests 1 - 6)	7 - 8
Graphs (Graph 1 - 5)	9
Section 4: Limitations	10
Limitations	10
Section 5: Conclusion	11
Results	11
Section 6: Resources	12
Sources	12

Section 1: Introduction

Project Description

The goal of this project is to write a report and design a system that has a load balancer send the requests for a web application that connects to a database. The target web application is a project in the database laboratory which will read, write, and update to a central MySQL server. There is a Tomcat server which will be the load balancer for multiple Apache instances (all deploying the same web application). The test application will read, write, and update to a MySQL virtual machine and report on the performance and success of each test.

System Architecture



The image above is a high level layout of our system. The automated requests are created by stress tests using Selenium to send the requests to the application. All of the requests are sent to the same URL which is the Apache server that is running HAProxy. HAProxy balances the requests in a round robin format across a number of Tomcat Instance that are running the same web application. From there, each web applications communicates with a central database running MySQL.

Programs Used

Vagrant [1]

The service that we used to create the virtual machines for all services in Vagrant. Vagrant is an open source software that is a tool for building complete development environments. For our project we used Vagrant to provision our multi-tier web application across a number of virtualbox instances with each with the operating system Centos 6.1.

Apache Tomcat [2]

The service that we used to deploy the web application and HAProxy is Apache Tomcat. This is an open source software that has built in logging and automatic deployment of war projects which is compressed web applications.

Google Web Toolkit [3]

The service that we used to develop the web application is Google Web Toolkit (GWT). GWT Plugin for the Eclipse IDE works well to create a web application in Java and be able to compile the project with JavaScript code. Once the project is compiled, a script is made to compress the compiled project into a war folder. The web application has customized logging that appends the results to a specified file. The results contain the IP address, the time, the number of read, writes, updates, duration, and overall status of the test. Furthermore, the web application makes separate asynchronous calls for each specified read, write, and update.

MySQL [4]

The service that we used to manage our test data is MySQL. The MySQL database is a relational database management system (DBMS). The test data in the database was five columns that are VARCHAR(200). Each row is unique with auto-increment primary key. This allows for more test data to be added without the worry of conflicting rows. The test results were 2.5 million rows.

HAProxy [5]

The service that load balanced the requests to the web application is HAProxy. HAProxy, which stands for High Availability Proxy, is an open source software TCP/HTTP Load Balancer and proxying solution. We used it collect all requests on a single front-facing apache server, and then distribute them to one or more web application servers in a round-robin fashion.

Selenium [6]

The service that we used to stress test the system is Selenium. Selenium is a tool use for browser automation. Primarily, it is for automating web applications for testing purposes. For the project we used the Selenium WebDriver, which is a collection of language specific bindings used to drive a browser. Our tests were written in java and we chose to automate Firefox. We created a runnable jar that took in 6 arguments (number of threads, number of requests per thread, number of reads, number of writes, number of updates, output file, and wait time for gwt to load). The addition of the runnable jar allows for fast creation and execution of tests.

Section 2: Testing Procedure

The variables that were tested are the number of Apache Tomcat instances, the number of reads per http request, the number of writes per http request, the number of updates per http request, and the number of requests per Tomcat instance. Lets say respectively each of the above factors are v-z. Each testing procedure scales one factor. This allows for a control case and only testing one variable at a time tells which test causes what errors.

Section 3: Results

Number of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	5	5	5	2	135	1544
40	5	8	5	5	5	2	202	1402
80	5	16	5	5	5	2	525	1374
160	5	32	5	5	5	2	1038	1407
320	5	64	5	5	5	2	2035	1303

Number of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	5	5	5	1	115	1371
40	5	8	5	5	5	1	229	1371
80	5	16	5	5	5	1	449	1312
160	5	32	5	5	5	1	838	1486
320	5	64	5	5	5	1	1762	1243

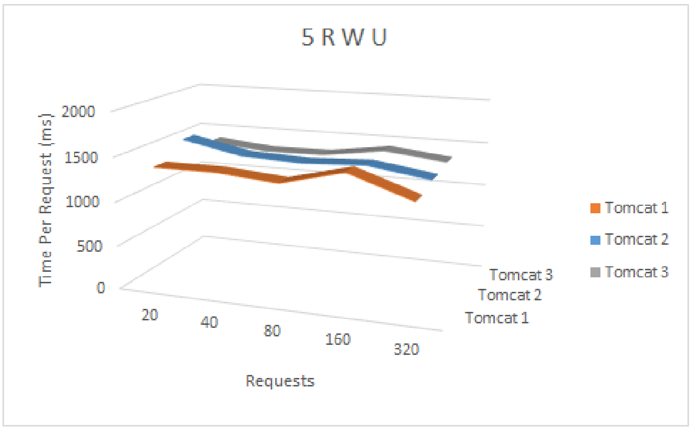
Number of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	5	5	5	3	134	1383
40	5	8	5	5	5	3	266	1318
80	5	16	5	5	5	3	525	1324
160	5	32	5	5	5	3	1035	1424
320	5	64	5	5	5	3	2048	1345

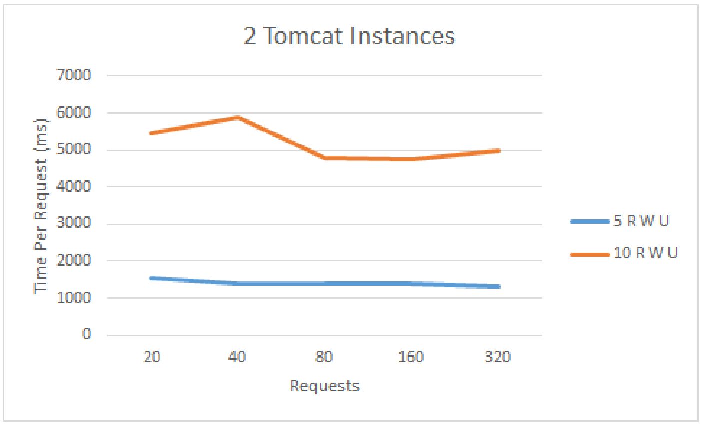
Results Continued

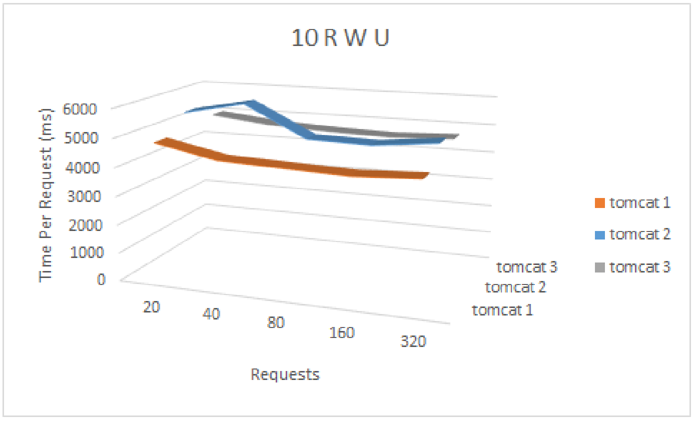
Num ber of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	10	10	10	2	135	5459
40	5	8	10	10	10	2	271	5892
80	5	16	10	10	10	2	525	4805
160	5	32	10	10	10	2	1050	4755
320	5	64	10	10	10	2	2115	4988

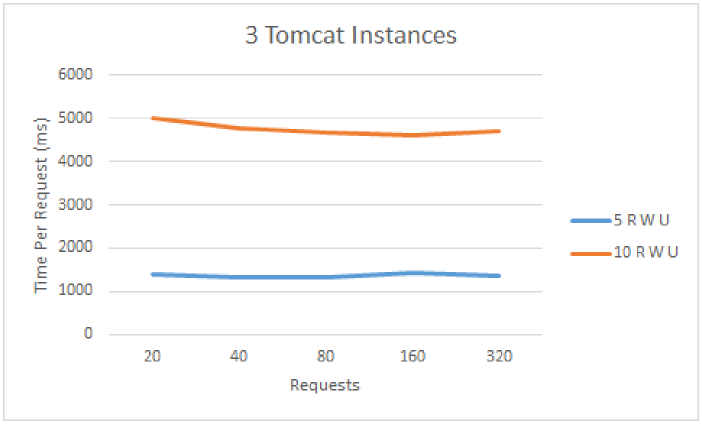
Number of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	10	10	10	1	117	4801
40	5	8	10	10	10	1	234	4361
80	5	16	10	10	10	1	469	4280
160	5	32	10	10	10	1	911	4207
320	5	64	10	10	10	1	1829	4310

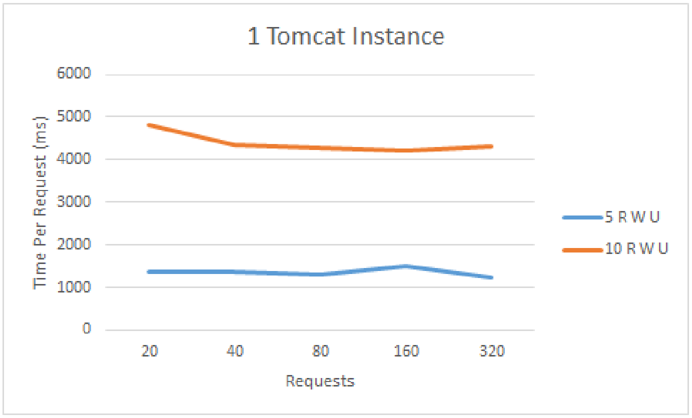
Number of Requests	Number of Threads	Number of Requests Per Thread	Number of Reads	Number of Write	Number of Updates	Number of Tomcat Instances	Stress Test time (seconds)	Results (Time per request in ms)
20	5	4	10	10	10	3	135	4994
40	5	8	10	10	10	3	266	4760
80	5	16	10	10	10	3	524	4658
160	5	32	10	10	10	3	1039	4603
320	5	64	10	10	10	3	2065	4555


5 R W U
Time Per Request (ms)
2000
1500
1000
500
0
20
40
80
160
320
Requests
Tomcat 3
Tomcat 2
Tomcat 1
Tomcat 1
Tomcat 2
Tomcat 3


2 Tomcat Instances
Time Per Request (ms)
7000
6000
5000
4000
3000
2000
1000
0
20
40
80
160
320
Requests
5 R W U
10 R W U


10 R W U
Time Per Request (ms)
6000
5000
4000
3000
2000
1000
0
20
40
80
160
320
Requests
tomcat 3
tomcat 2
tomcat 1
tomcat 1
tomcat 2
tomcat 3


3 Tomcat Instances
Time Per Request (ms)
6000
5000
4000
3000
2000
1000
0
20
40
80
160
320
Requests
5 R W U
10 R W U


1 Tomcat Instance
Time Per Request (ms)
6000
5000
4000
3000
2000
1000
0
20
40
80
160
320
Requests
5 R W U
10 R W U

Section 4: Limitations

When creating this stress-test experiment, we faced two primary limitations: processing power available and consequently the time cost of each test run. We originally intended to make use of Amazon Web Services (AWS), but we were unable to figure out how to use it in the timeframe that we had for our project. We instead opted to run the experiment locally on a single machine with 8GB of RAM and a i5 core processor. This meant that all of the virtual machines (load-balancer, tomcat instances, and database) were run on the same machine and had to share the same resources. On top of that, The requests that were generated by the stress-testing program were generated on the same machine as well. With that being said, adding more tomcat instances increased the number of servers available for the load-balancer to round-robin requests to, but it also divided up the amount of resources available to each virtual machine into smaller and smaller pieces.

The second major limitation we had in conducting this experiment was the time cost of each test. As shown in the data, the shortest test (5 threads at 4 requests each for 20 total requests) took approximately 2 minutes and 15 seconds, and as we doubled the number of requests per thread, the time cost of the test followed suit. What this means is that the time cost grew exponentially. Due to this limitation we were not able to run tests with the number of requests that we were hoping for. We were also unable to run test cases more than once, which would have helped smooth out any anomalies or outliers in our data.

Section 5: Conclusion

The results show that connection pooling decreases the wait time for the average user. The method of connection pooling allows the connection to the database to be cached for later users. This allows for the average wait time to decrease with more requests because the initial connection only occurs once. The results of our experiment were inconclusive for tomcat instances decreasing wait time. We were unable to show that increasing the number of Tomcat instances would correlate with a performance gain. We believe that we arrived at this result due to our hardware limitations. We were unable to test the system with enough requests to reach its true limits. Also, the exponential time cost of the tests were especially prohibitive and prevented us from collecting more than one set of data for each test case.

We believe that with more processing power, we would be able to run our experiment at stress-levels that better reflected the real world and would be able to do so in significantly less time. The environment we performed the test on simply did not have the capability to handle (and simulate simultaneously) the number of requests we had originally intended for this experiment. Theoretically, it would make sense that more tomcat instances would spread the workload and prevent any one instance from becoming overwhelmed with requests, improving performance. However, this model makes the assumption that every Tomcat instance has enough resources available to it in order to run as efficiently as a single standalone Tomcat instance. This was not the case in our experiment as we were limited to a single processor and 8GB of RAM. Had we used a service such as AWS where our resources would expand and contract to meet our needs, we would have been able to achieve this ideal model of **n** Tomcat instances running with all the resources they need. However, the learning curve associated with Amazon AWS made it an unviable option for this project.

That being said, the setup of our project is entirely automated, meaning that if someone who did have adequate hardware to perform this experiment (such as Amazon AWS) wanted to, they would be able to with minimal effort. All that is required to setup and run our project is basic vagrant and Java knowledge. Furthermore, we have logging setup so that they would be able to find the root cause of any crash during their experiments. They would even be able to parse the web application logs to get statistical data on the request times and failures using one of the scripts included in our project.

If someone wanted to use our project to run their own custom HTTP request stress-testing, they would be able to create their own with some simple bash scripting as our stress-test program is a JAR that takes seven parameters (requests per thread, number of threads, number of reads/writes/updates, log file, and request duration).

Section 6: Resources

Sources

1. Vagrant https://www.vagrantup.com/
2. Apache Tomcat http://tomcat.apache.org/
3. Google Web Toolkit http://www.gwtproject.org/
4. MySQL http://www.mysql.com/
5. HAProxy http://www.haproxy.org/
6. Selenium http://docs.seleniumhq.org/projects/webdriver/