SENG 426

Software Quality Engineering

Summer 2021

University of Victoria

Delivery 4: Performance and Scalability Testing

July 19th, 2021

David Bishop	V00884250
Amaan Makhani	V00883520
Ben Austin	V00892013
Nolan Kurylo	V00893175
Liam Franke	V00887604
Evan Roubekas	V00891470

Table of Contents

Task I - Load Testing

Test Configuration

The load test executes the below actions:

1) Navigate to the accounts page 100% of the time
2) Request a new account type 2% of the time
3) Navigate to the transaction page 100% of the time
4) Create a new transaction 50% of the time such that if the chequing account has a balance of more than $50.00, transfer $50.00 from the chequing account to the savings account, otherwise transfer 100% of the money on the Saving account to the chequing account

To execute the above actions a few tools were used. The console interface through the web browsers inspect menu was used. This was used to obtain the backend api calls used for corresponding actions on the website. All calls were made to simulate real user behaviour, this means small requests needed for the UI to present options were all executed. This was intended to better simulate the volume of requests the applications needs to support.

To control the frequency of actions throughput controllers were used. The percent of threads to execute the nested action was then entered. To determine what create transaction request was to be made two if controllers were used. One controller tested for above 50 in the checking balance while the other was checking for below or equal to 50 as a balance.

The checking account number, saving account number, and checking balance was all obtained through a previous response and saved using the JSON decoder. In the decoder multiple paths and variables were entered in order to store the response data. These variables were then used to complete the create transaction request.

Think times were added between tasks to simulate user behavior. To set these our group completed the actions as fast as possible and set the base time using this value. Then we estimated the random addition time to support all ranges of users.

To ensure all users logged in once and had their own account we replaced the users.csv with new data generated through a script. We then read in the username and supplied our constant password to the authenticator in place of the username and password values.

To supply the same content type we used the header manager and set the content type here to application/json. To ensure we didn't have to enter the base url we used request defaults and set this to the needed URL path. This allowed us to change the URL with ease if needed.

To execute the demanded load the thread group was set up to create 200 users. Each user is added every one second. This means the total ramp up period is 200 seconds. These threads were to run infinite times for an hour in order to achieve our results.

Calculated Values, Charts and Statistics

The following section addresses all questions outlined in the lab manual and rubric. Each question analyzes the behaviours of the system as well as any implications that may be known to be causing issues with the system's performance. View Appendix A for all charts, graphs, and figures collected that are referenced in the following section. The charts and graphs were obtained after running the test in non-GUI mode and generating an HTML report from the .jtl file in jmeter.

1) Average, min and max response times per action (sampler) and aggregate

As seen in Figure A.4, each of the actions are aggregated into average, minimum, and maximum response times in milliseconds. Overall, the average response times of all actions was found to be 748.21 ms, with all minimum response times of 6 ms, and all maximum response times of 189927.

2) A chart of the response times over time per action (sampler) and aggregate

In Figure A.1, the response times over time for each action is displayed. It is noticeable that the largest response times occurred after approximately 16 minutes of test execution; a spike in response times occurred. This could be due to a constraint on the system such as the CPU or memory being overloaded.

3) The error rate per action (sampler) and aggregate

Figure A.4's "Execution" column outlines each action's error rate as a percentage. Every sampler was seen to have a 0% error rate except for the "Create New Transaction Request" sampler. This was due to how the system's cookie expirations were set up. After a certain amount of time, the virtual user was no longer authenticated as it's cookie had expired. This resulted in "unauthorized" responses for the request, requiring the cookie's expiration to be reset.

4) A chart of the response codes over time

Figure A.2 demonstrates the system's throughput of response codes over the period of test execution. It can be seen that the error response codes (400) were kept to a minimum throughout the testing. Furthermore, the successful 201 response codes where stable all

throughout while the 200 response codes were slightly less stable, having a larger fluctuation in the number of responses per second.

5) A chart of the number of active threads over time

Figure A.3 shows that the load test consisted of 200 generated virtual users for each of the 2 slave machines that were set up. The graph is segmented based on the two unique machines to show that 200 virtual users corresponded to each machine.

6) Average, min, and max CPU usage

Figure A.5 shows the cpu usage during the load test. Throughout the load test we can estimate an average cpu usage of 20, with a high of just under 36, and a low of 15 (ignoring the first low made during startup). The non-linearity and high fluctuations of these values goes to show that no bottleneck is currently present. If it were, the graph would stabilize and flatten.

7) Average, min, and max memory usage

Figure A.6 shows the memory usage during the load test. Throughout the load test we can estimate an average memory usage of 90.1, with a high of 90.2, and a low of 90. Since the application is under high load for this test, it is expected that a large amount of the memory will be taken up. It makes sense that the memory usage remains consistent throughout the entirety of the test, as the high workload demands the system to constantly be writing to and retrieving from the memory of the system.

Software Availability

Using the aggregate model:
Availability = Successful Request / Total Requests
= (Total Requests - Failed Requests) / Total Requests
= 98.786541075 %

Average System Throughput

Think Times:
Before Requesting New Account: 5 seconds
Fill in New Account form: 4 seconds
Before Retrieving Transactions: 2.5 seconds
Decide to make a transaction: 1.5 seconds
Fill in Transaction Form: 3 seconds

Z = Average Think Time: (5 + 4 + 2.5 + 1.5 + 3) / 5 = 16 / 5 = 3.2 seconds
N = Number of Users: 400 (200 virtual users for 2 slave machines)
R = Average Response Time: 748.21ms = 0.74821 sec

Using Response Time Law: $R = N/X_o - Z$

X_o = N / (R+Z)
= 400/(3.2 + 0.74821)
= 101.312 req/s

Throughput Bound Law

X_1 = 22.8 tps
X_2 = 11.38 tps
X_3 = 0.46 tps
X_4 = 11.38 tps
X_5 = 0.46 tps
X_6 = 11.38 tps
X_7 = 0.46 tps
X_8 = 22.77 tps

Default JMeter PacketSize = 4096 bytes
Bandwidth = Network Received + Network Sent (KB/sec)

S_i - Service Time
S_i = packet size / bandwidth
S_1 = (4096 * 8 bits) / [(30.31 + 5.13 KB/sec) * (8000 bits/1 KB)] = 0.116 seconds
S_2 = (4096 * 8 bits) / [(10.17 + 3.77 KB/sec) * (8000 bits/1 KB)] = 0.294 seconds
S_3 = (4096 * 8 bits) / [(0.71 + 0.09 KB/sec) * (8000 bits/1 KB)] = 5.12 seconds
S_4 = (4096 * 8 bits) / [(14.79 + 2.33 KB/sec) * (8000 bits/1 KB)] = 0.239 seconds
S_5 = (4096 * 8 bits) / [(0.46 + 0.09 KB/sec) * (8000 bits/1 KB)] = 7.45 seconds
S_6 = (4096 * 8 bits) / [(8.61 + 2.29 KB/sec) * (8000 bits/1 KB)] = 0.376 seconds
S_7 = (4096 * 8 bits) / [(0.26 + 0.41 KB/sec) * (8000 bits/1 KB)] = 6.11 seconds
S_8 = (4096 * 8 bits) / [(86.04 + 4.74 KB/sec * (8000 bits/1 KB)] = 0.0451 seconds

Utilization Law
U_i = $X_i * S_i$

U_1 = $X_1 * S_1$
= 2.6448

U_2 = $X_2 * S_2$
= 3.3457

U_3 = $X_3 * S_3$
= 2.3552

U_4 = $X_4 * S_4$

= 2.7198

U_5 = $X_5 * S_5$
= 3.427

U_6 = $X_6 * S_6$
= 4.2789

U_7 = $X_7 * S_7$
= 2.8106

U_8 = $X_8 * S_8$
= 1.0269

Service Demand Law
$D_i = U_i / X_o$
X_o = 101.312 req/s

D_1 = U_1 / X_o
= 0.0261

D_2 = U_2 / X_o
= 0.03302

D_3 = U_3 / X_o
= 0.02324

D_4 = U_4 / X_o
= 0.02684

D_5 = U_5 / X_o
= 0.03383

D_6 = U_6 / X_o
= 0.04223

D_7 = U_7 / X_o
= 0.02774

D_8 = U_8 / X_o
= 0.01013

Throughput Bound Law
$X_{max} <= \min[1/D_{max}, N/\text{sum}(D_i)]$

X_{max} <= min[1/0.04223, 400/0.22313]

X_{max} <= min[23.68, 1792.68]

X_{max} <= 23.86 tps

→ The maximum achievable system throughput of the system is 23.86 tps.

Analysis of System Behaviour

In terms of errors, our Load Testing only produced a single error code from the same request. Of the 291316 requests executed, 3535 of them failed, giving us an error rate of 1.21%. Every single failure was a result of the Create New Transaction Request, and resulted in a 400/BadRequest being returned to the server. When comparing this to the results of our stress testing, it is notable that we had 7 different types of errors, from 7 different requests, versus the single error from a single request in our Load Testing. As discussed previously, this behaviour occurs as the Create New Transaction Request relies on cookies, and if the request takes too long the cookie may expire, or have been overwritten if the workload on the system is too great at one time. Therefore, if this occurs, the user trying to Create a New Transaction Request will be unauthorized to do so, resulting in a failure. These errors most likely occurred as the number of virtual users increased the amount of requests being made increased at a larger multiplicity, leading to an overbearing workload on the system.

The Response Codes Per Second offer a full scope working glimpse of our system, and is very promising. For the entirety of our test, we were averaging around 75 response codes 200's per second, 12 response codes 201's per second, and 1 response code 400 per second. A code of 200 and 201 indicates a request successfully being fulfilled, while the cause for the 400's is discussed above. This can be seen in figure A.2.

Task II - Stress Testing

Test Configuration

The stress test used the same test plan described above. The difference was the thread group settings. The number of users was set to start at 500. The users were added every 2 seconds. This gave us a ramp up time of 1000 seconds. The thread timeout was also set to timeout after twenty seconds. The number of threads and ramp up time was then changed as described below.

To start, our team developed a Python script that populated the .csv files in the neptune bank with 500 user accounts, checking and saving accounts, account balances, customer logins, and other associated information. These new .csv files successfully populated the h2 database for the application when it was launched in development mode.

A test environment was set up with three machines on the same local network. One acting as the master, one acting as the slave, and one acting as the target. The master machine ran jmeter and executed the test plan pointing at the slave machine. The slave machine ran the perfmon server agent and forwarded requests to the target machine, which was running a development version of the Neptune Bank application. To correctly set up the test environment, we had to configure the jmeter.properties file to disable SSL due to being on a local network, and add the required IP addresses and ports to the file.

After correctly configuring the properties files on the machines, the jmeter test plan was modified to point at the machines on the network. Next, the application was started on the target machine, the perform server agent was started on the slave, and finally, the jmeter test was started on the master.

Calculated Values, Charts and Statistics

The following section addresses all questions outlined in the lab manual and rubric. Each question analyzes the behaviours of the system as well as any implications that may be known to be causing issues with the system's performance. View Appendix B for all charts, graphs, and figures collected that are referenced in the following section. The charts and graphs were obtained after running the test in non-GUI mode and generating an HTML report from the .jtl file in jmeter.

1. How many concurrent users are necessary to increase the average response time to more than 2 seconds?

As you can see in figure B.3 the first time (other than during startup at time 0) the response time takes over 2 seconds and is just past 15 minutes and 30 seconds. Since our ramp up time is 2 seconds for each new user we know that at that time we had 465 concurrent users.

2. How many concurrent users are necessary to increase the error rate to more than 5%?

Throughout the ramp up of users we got a 1.01% error rate from 500 concurrent users (figure B.4). Extrapolating this data we can estimate that we would reach a 5% error rate at 2500 users.

3. How many concurrent users are necessary to degrade the system's performance enough so it becomes unusable for the majority of users? Can this even be achieved, and if so, how does the system behave in that scenario? Does it enter into an

unrecoverable state, hangs or crashes, or does it return to normal after the stress is reduced?

In the response codes per second image (figure B.5) you can see that around 15 minutes and 45 seconds the response codes per second drop off precipitously, this is a good indication that the systems performance has degraded to the point of unusability. The behavior of the system is a slower time to respond to requests and an increased number of failed (400 code) requests. At this time with a ramp up of a new concurrent user every 2 seconds the system would have had 473 concurrent users. Since this stress test was only run for a certain period of time and number of users and the unusable state was only entered near the end we can only predict that the system would recover if the load was lifted.

4. Do you believe the system's performance allows it to be released into production? Consider the hardware, network, user base vs. peak concurrent users, variations in user behavior, performed actions, think-time etc

Yes based on the combination of different metrics we gathered (memory usage, cpu usage, response time, etc) we are confident that this app is ready for production, past a certain number of concurrent users the site does become unusable but for there to be that many concurrent users is unrealistic and would be an extraordinary event.

Performance Degradation

When running the stress test, multiple attempts were executed in hopes of achieving degradation of the system. To do this, the attempts consisted of increasing the number of virtual users. In result, the error percentage of our system increased slightly greater than 1%; degradation could not be achieved. In several more attempts, the user think times were adjusted so that they were lower (less realistic) than normal. Even so, nearly the same result was achieved with no degradation. It can be concluded that the system seems to respond well to extraneous stress as the stress test could not successfully wear down the application.

Analysis of System Performance

Response Codes Over Time

Throughout the duration of the stress test, the number of responses per second for each of our 3 response codes remain very consistent over the course of the test running, as demonstrated in Figure B.3. There is a spike that occurs from the 15:30 mark to the 15:45 mark, where an average of about 200 successful response codes per second and 30 successful with resource added codes per second were being sent. After this spike, both of these areas fell to stagnant around 50 responses/sec and 7.6 responses/sec respectively. The amount of failures remained

constant over the entirety of the test, ranging from 0 - 2.5 failed responses per second. Most notably, at the 16:13 and the 16:23 mark all response codes per second fell to zero, before returning to normal. There are a few explanations for these sudden peaks and valleys. Firstly, the initial jump that sees all response codes per second being returned at around 5 times their normal rate could be due to all accumulated requests that have been waiting for response to finally be fulfilled. Lastly, in the times when the returned response codes drop to zero, it could be due to the system being stuck waiting on a single request for a moment and not completing any others.

Response Times Over Time

The results of the stress test depict how different requests made throughout the duration of the test's execution have variable response times. The graph in Figure B.4 shows that as the number of users increases, the response times on almost all requests stay at a reasonable, and consistent level. Taken in combination with Figure B.7, it is clear that the majority of requests are responded to in less than 500ms, but occasional spikes do occur, as seen in Figure B.4. There are 3 notable areas where the response times are noticeably higher, from 15:45 - 15:50, 16:10 - 16:15, and 16:20 - 16:25. In each of these areas, the peak response times are 76585ms, 91176ms, and 226135ms respectively. These enormous jumps in response times are most likely caused by the system being overwhelmed by the workload, and ignoring a request for an extended period of time as it is working through other requests, as a result of more virtual users being created and requests being made.

Appendix A

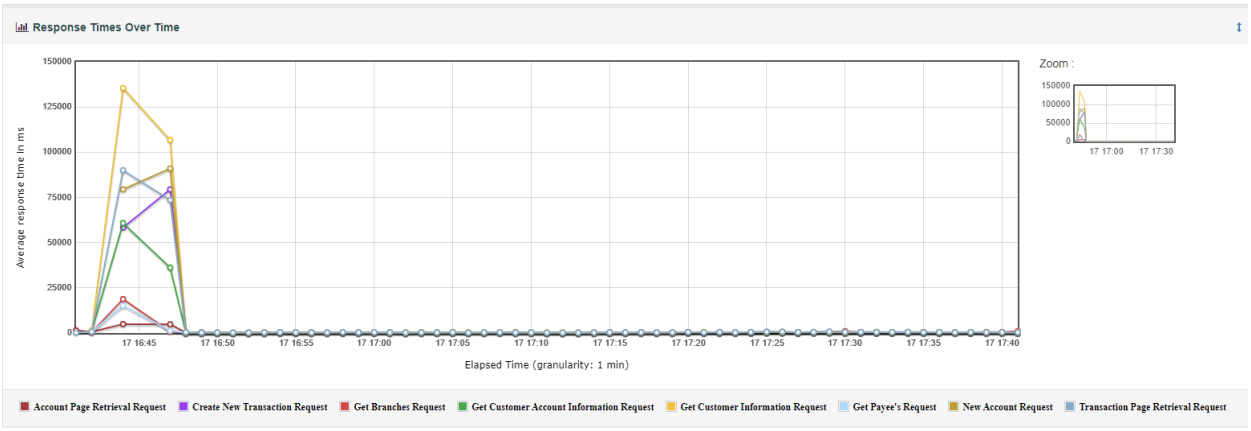

Figure A.1 - Load response over time

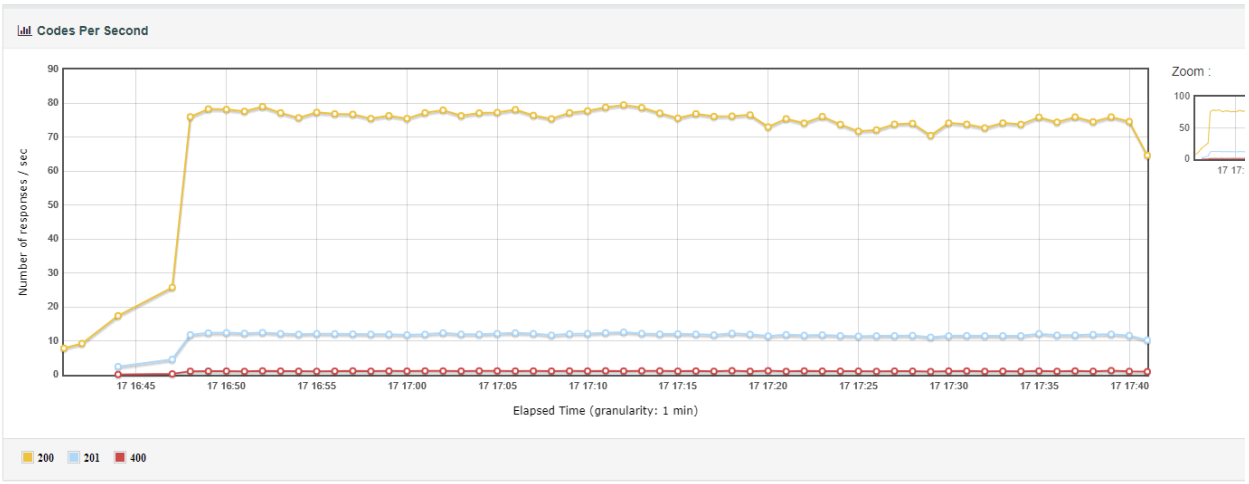

Figure A.2 - Load codes per second

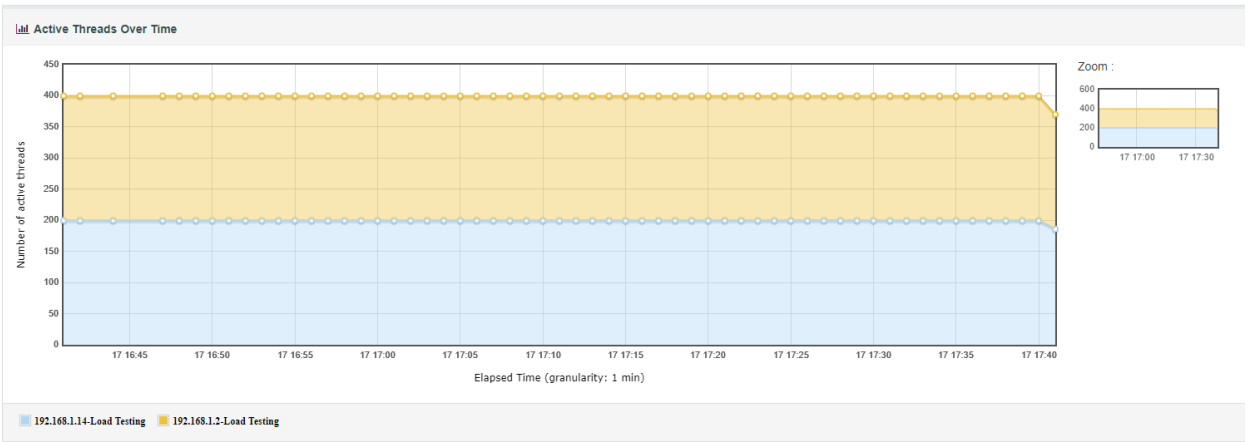

Figure A.3 - Load active threads over time

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**291316**	**3535**	**1.21%**	**748.21**	**6**	**189927**	**206.00**	**678.00**	**1179.95**	**2679.83**	**80.91**	**151.24**	**18.54**
Account Page Retrieval Request	82086	0	0.00%	306.06	7	185288	233.00	616.00	948.90	2299.94	22.80	30.31	5.13
Create New Transaction Request	40768	3535	8.67%	1087.74	9	185341	289.00	839.00	1365.00	2809.98	11.38	10.17	3.77
Get Branches Request	1636	0	0.00%	374.78	7	139650	158.50	505.60	920.15	2519.15	0.46	0.71	0.09
Get Customer Account Information Request	40860	0	0.00%	808.51	6	183060	253.00	760.00	1221.00	2534.95	11.38	14.79	2.33
Get Customer Information Request	1636	0	0.00%	1549.08	9	181371	197.00	682.30	1242.35	4068.48	0.46	0.46	0.09
Get Payee's Request	40856	0	0.00%	328.80	6	183631	211.00	595.00	909.00	2140.98	11.38	8.61	2.29
New Account Request	1629	0	0.00%	1225.97	8	183105	189.00	682.00	1191.00	2756.80	0.46	0.41	0.14
Transaction Page Retrieval Request	81845	0	0.00%	1183.73	9	189927	299.00	1000.00	1655.95	3485.96	22.77	86.04	4.74

Figure A.4 - Load statistics

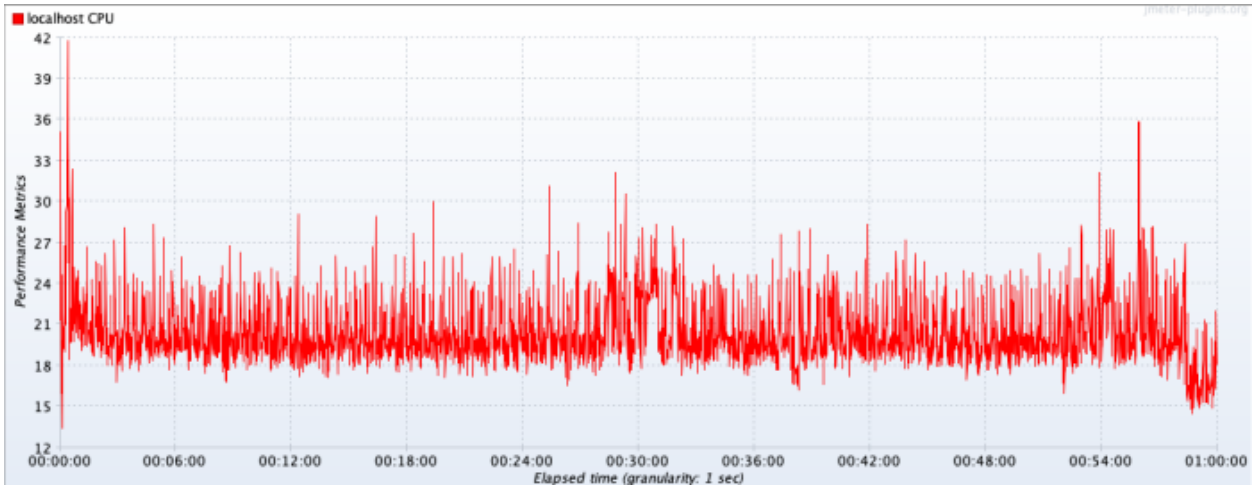

Figure A.5 - Load CPU Usage

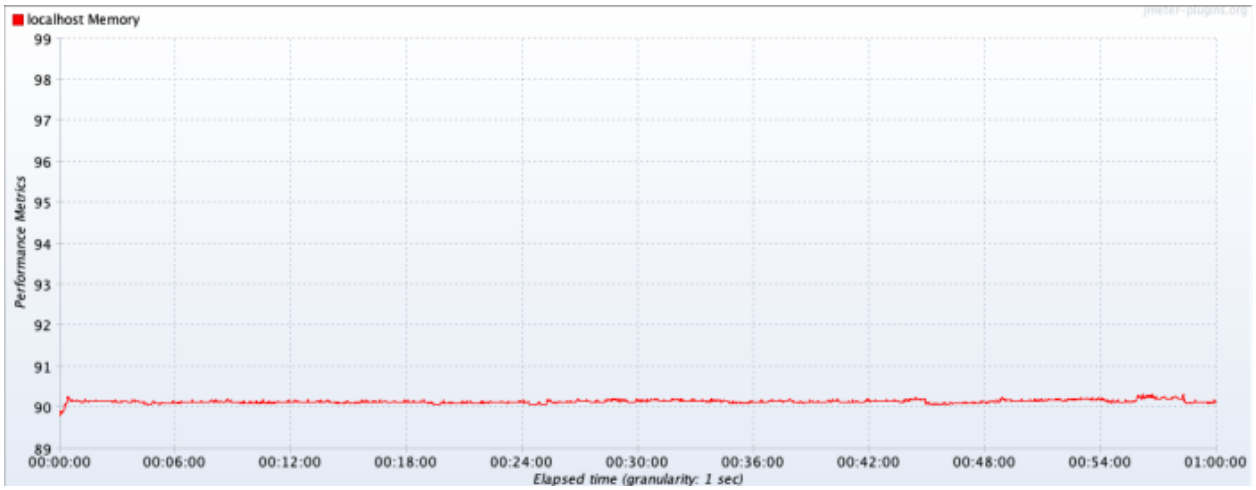

Figure A.6 - Load Memory Usage

Appendix B

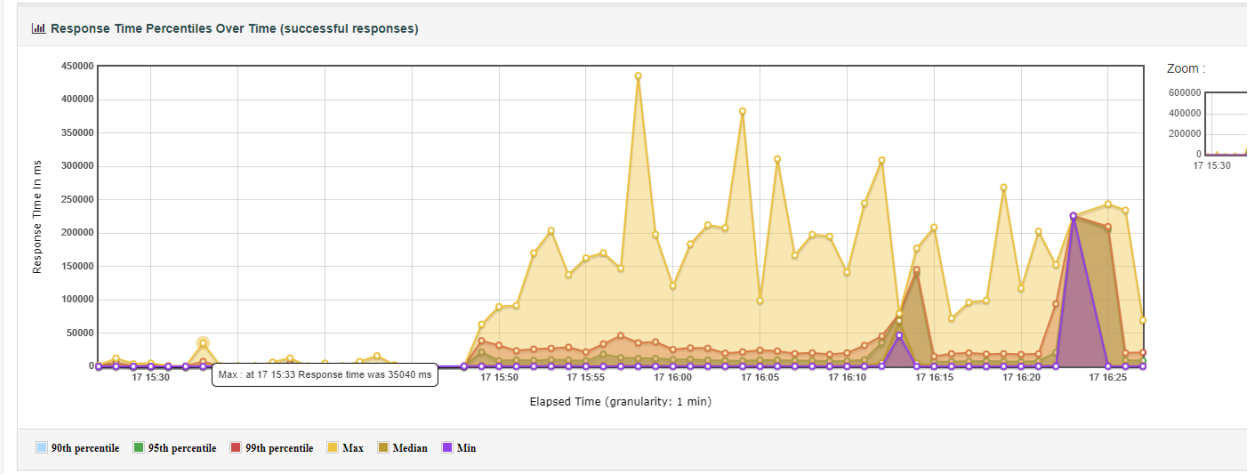

Figure B.1 - Stress response time percentiles

Errors

Type of error	Number of errors	% in errors	% in all samples
400/Bad Request	3641	86.12%	1.01%
Non HTTP response code: org.apache.http.conn.HttpHostConnectException/Non HTTP response message: Connect to 192.168.1.3:4080 [/192.168.1.3] failed: Operation timed out (Connection timed out)	503	11.90%	0.14%
Non HTTP response code: java.net.SocketException/Non HTTP response message: Network is unreachable (connect failed)	54	1.28%	0.01%
Non HTTP response code: org.apache.http.NoHttpResponseException/Non HTTP response message: 192.168.1.3:4080 failed to respond	21	0.50%	0.01%
500/Internal Server Error	7	0.17%	0.00%
Non HTTP response code: java.net.SocketException/Non HTTP response message: Connection reset	1	0.02%	0.00%
Non HTTP response code: java.net.SocketException/Non HTTP response message: Operation timed out (Read failed)	1	0.02%	0.00%

Figure B.2 - Stress errors

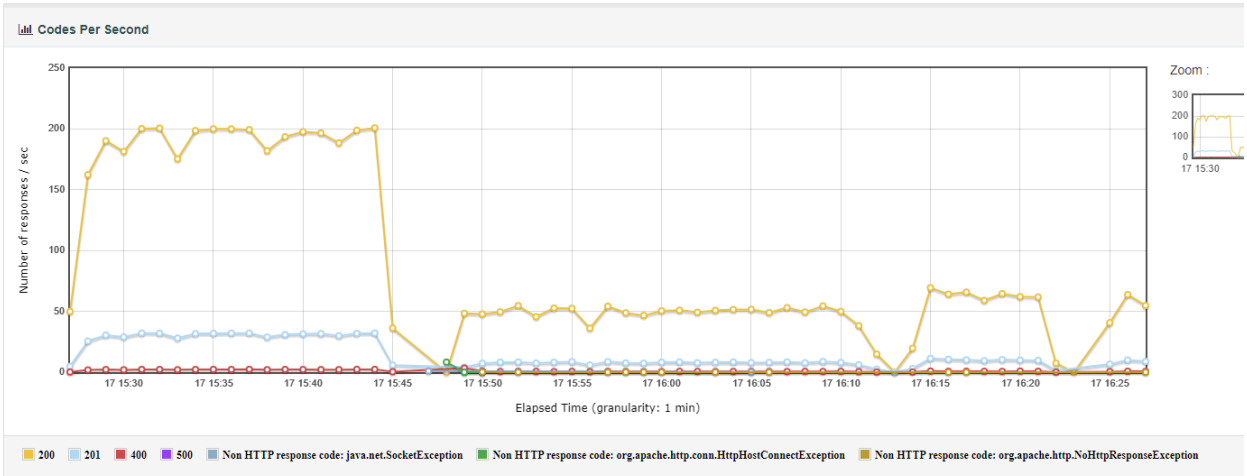

Figure B.3 - Stress codes per second

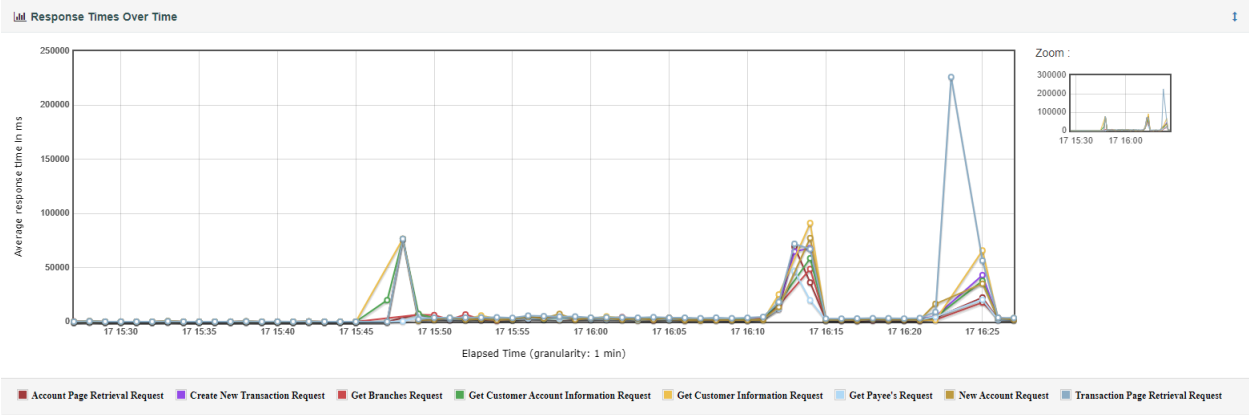

Figure B.4 - Stress Response Times Over Time

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**361291**	**4228**	**1.17%**	**1628.61**	**0**	**436572**	**1142.00**	**6083.90**	**11902.00**	**205878.98**	**100.10**	**180.89**	**22.90**
Account Page Retrieval Request	101763	26	0.03%	1230.10	0	212070	714.00	5221.90	8578.75	72682.84	28.20	37.06	6.34
Create New Transaction Request	50583	3789	7.49%	1729.04	0	214354	787.00	5439.90	9527.15	76575.98	14.05	12.62	4.64
Get Branches Request	2025	0	0.00%	1285.64	5	220518	37.00	2389.40	4228.00	19804.06	0.56	0.88	0.12
Get Customer Account Information Request	50688	75	0.15%	1641.20	1	309871	773.00	5696.80	10372.95	76558.00	14.06	18.27	2.87
Get Customer Information Request	2027	7	0.35%	2016.13	7	217076	49.00	2372.80	4809.80	32195.36	0.56	0.57	0.11
Get Payee's Request	50634	11	0.02%	1058.54	0	243815	644.00	4464.80	7315.90	27949.43	14.06	10.64	2.83
New Account Request	2024	15	0.74%	1728.29	1	203483	45.00	2585.50	5044.25	27102.50	0.56	0.51	0.18
Transaction Page Retrieval Request	101547	305	0.30%	2253.00	1	436572	1752.00	8257.00	16348.30	198085.81	28.16	100.56	5.85

Figure B.5 - Stress Statistics

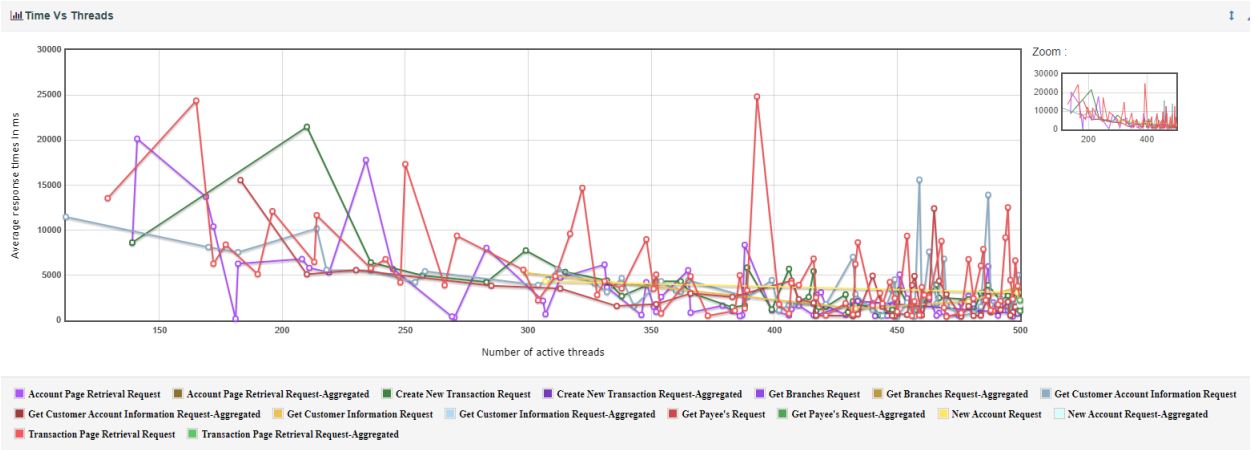

Figure B.6 - Stress Time vs Threads

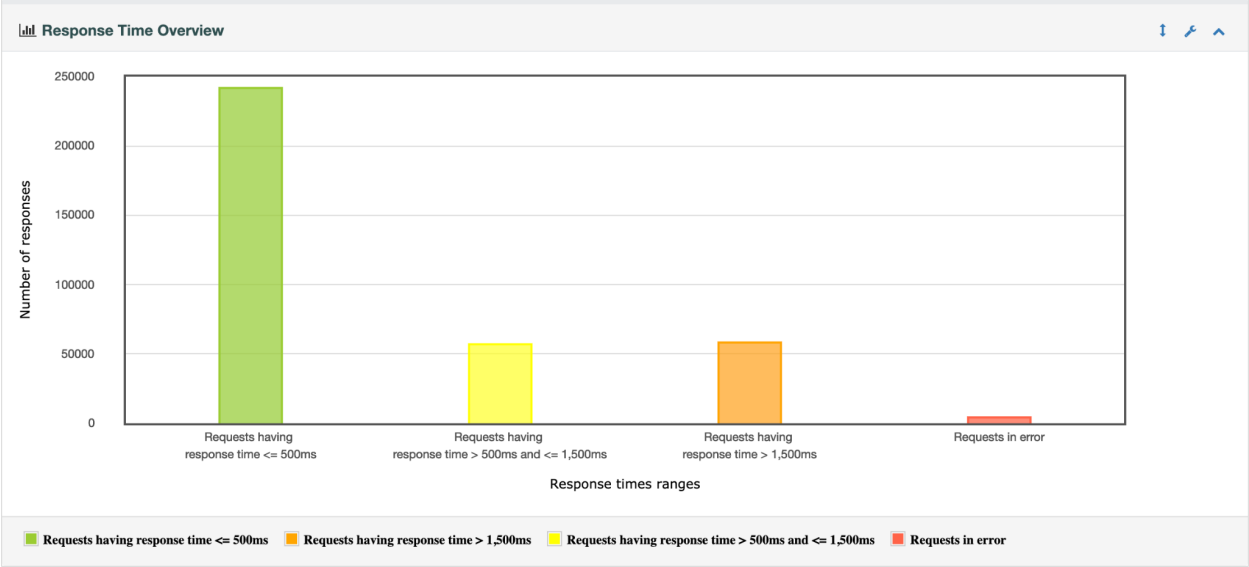

Figure B.7 - Response Time Distribution