CSE 330 Webserver Test

Server Info:

Operation System	Linux version 4.9.91-40.57.amzn1.x86_64
System bit number	ELF 64-bit LSB executable, x86-64, version 1 (SYSV)
CPU	CPU0: Intel(R) **Xeon**(R) CPU E5-2676 v3 @ 2.40GHz (family: 0x6, model: 0x3f, stepping: 0x2)
RAM	total used free shared buffers cached 993 801 191 0 144 342
Hard disk size	devtmpfs 488M 56K 488M 1% /dev tmpfs 497M 0 497M 0% /dev/shm /dev/xvda1 7.8G 3.4G 4.3G 45% /

Test Client info:

Operation System	System Version: macOS 10.13.4 (17E199)
System bit number	RELEASE_X86_64 x86_64
CPU	CPU: Intel(R) Core(TM) i5-5257U CPU @ 2.70GHz
RAM	8 GB 1867 MHz DDR3
Hard disk size	250.79 GB 78.94 GB 68%

Test Tool

Name	ApacheBench
Version	Version 2.3
Information	ApacheBench (ab) is a single-threaded command line computer program for measuring the performance of HTTP web servers. Originally designed to test the Apache HTTP Server, it is generic enough to test any web server.

Introduction

This Evaluation experiment is going perform on our web server. Our website is built by Django on EC2 instances. To improve the performance of Web servers, we have studied the Web server's performance, namely, the response time and throughput, which can be perceived by end users directly. The web server we use on our server is apache, apache is one of the most powerful server service in the world. For apache itself, we have many tools to optimize it to get a higher performance. Also, since Ec2 is provided by amazon, it actually provide many different types of instances with different properties, all these properties will influence the final performance of our website. Amazon provide some optimization tools like Elastic Load Balancing products to help AWS web server. So in this report, I am going to first evaluate the performance with and without optimizations. Then I am going to evaluate the web server performance with/without Elastic Load Balancing (ELB).

Experiment

Experiment environment setup

The test tool we are going to use is ApacheBench.
In order to receive the accurate result, I clean the cache every time before the test. So before the test, the server should have same free memory.

The test tool we are going to use is ApacheBench.
It is a really good tool to do the load test.
Pre-Test:
From the pre-test we found out that our server could only load 100 users at the same time when we try to load 10000 times total, so 100 will be the largest co-user number during our whole

Test ID	Test Case
1-1	Load 10000 times for the same page and load it with 10 users in the same time
1-2	Load 10000 times for the same page and load it with 25 users in the same time
1-3	Load 10000 times for the same page and load it with 50 users in the same time
1-4	Load 10000 times for the same page and load it with 75 users in the same time
1-5	Load 10000 times for the same page and load it with 100 users in the same time

evaluation test design.

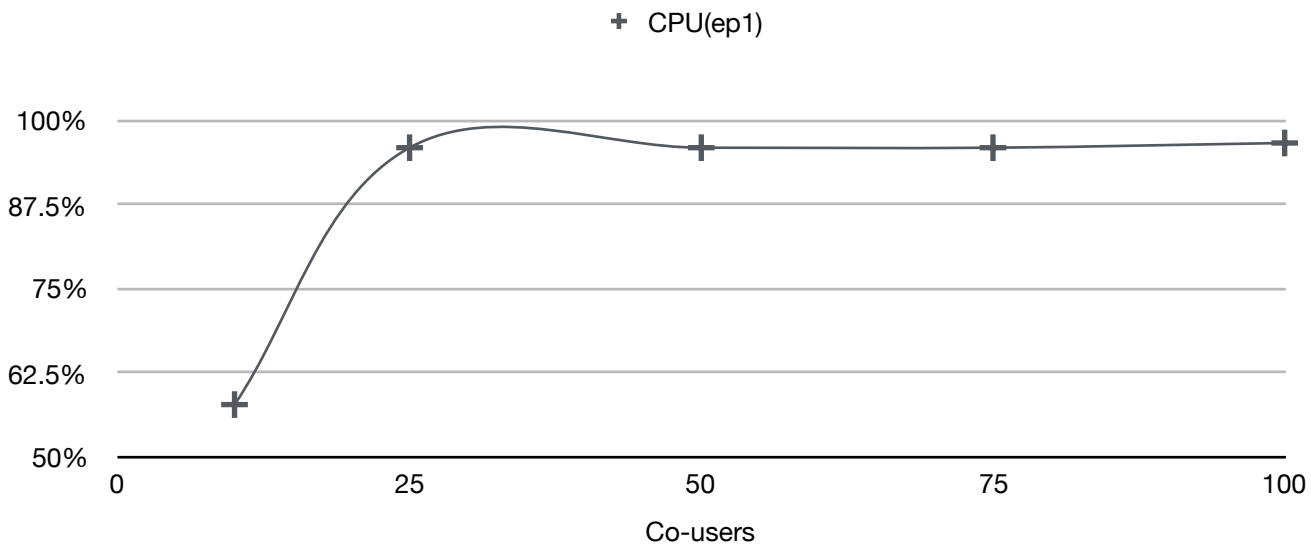
Experiment1 introduction

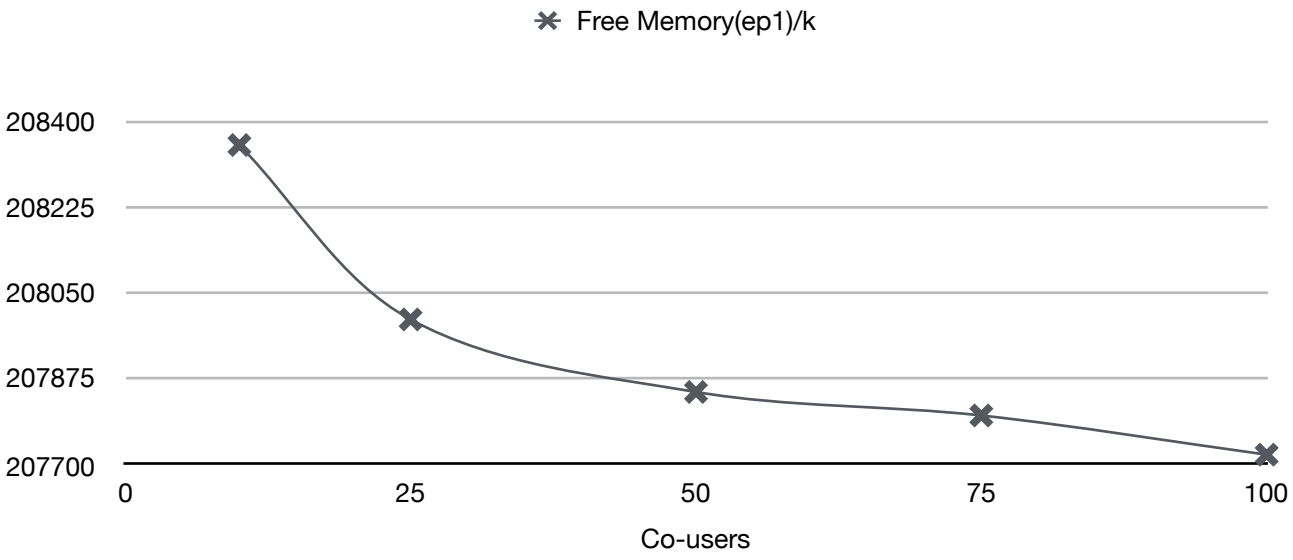
Load Test parameter:
CPU usage:
The usage of CPU display the performance of the instance. Being aware of the impact of your website on the server's CPU can help prevent any potential account suspensions for high resource usage.
Free Memory:
The larger the free memory means the smaller the RAM been used. So with a smaller usage of RAM means the data has been distributed stored. Also means that the web-server has a better performance.
Requests Per Second:
RPS, is a scalability measure characterizing the throughput handled by a system.
Time per request:
The average amount of time it took for a *concurrent* group of requests to process.
Time per request(mean):
The average amount of time it took for a *single* requests to process.
Transfer rate:
The higher the transfer rate, the better the website data transfer. Also means a better web server.
Request served time:(90%)
Data for 90% is a good way to show the serve performance for apache server.

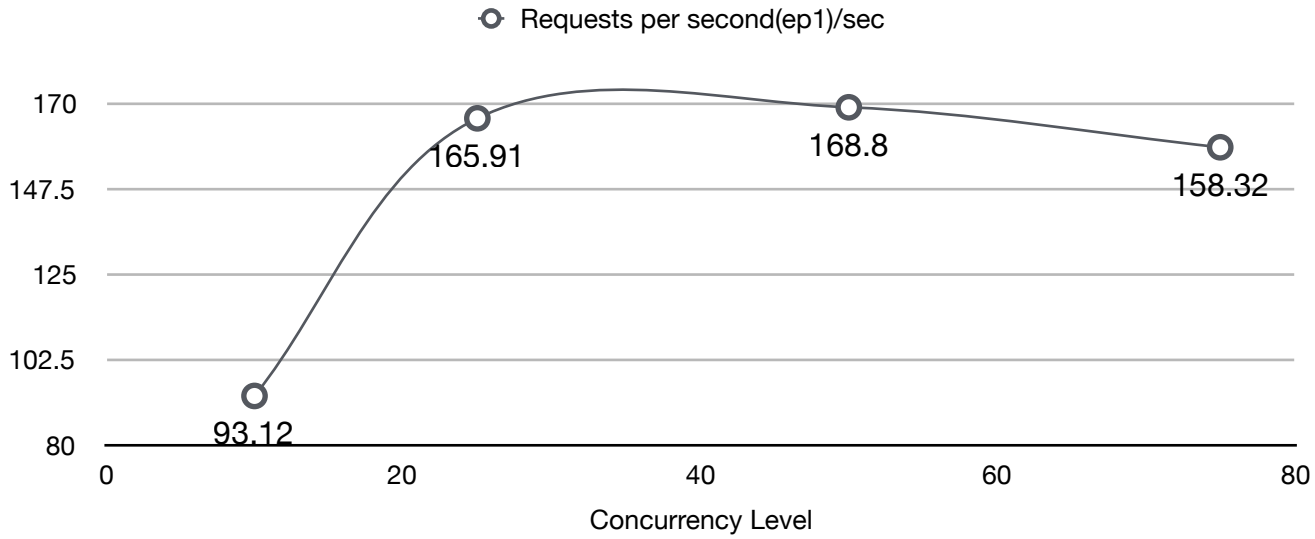
Experiment 1 Result

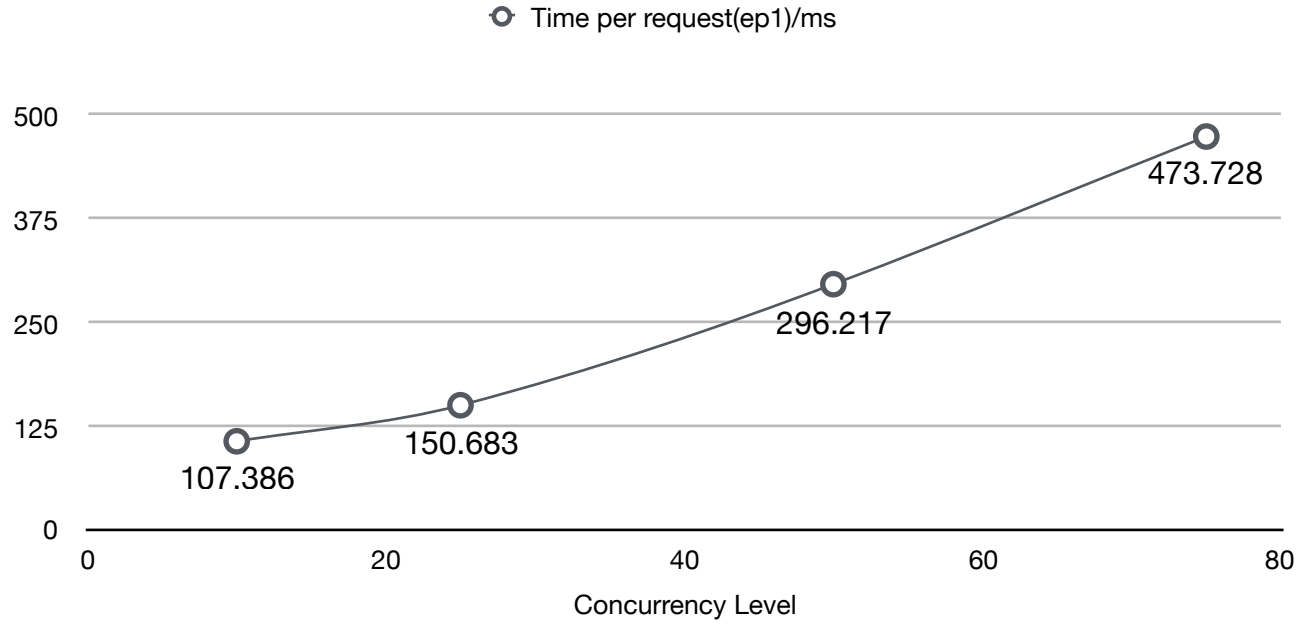
Basic Info:

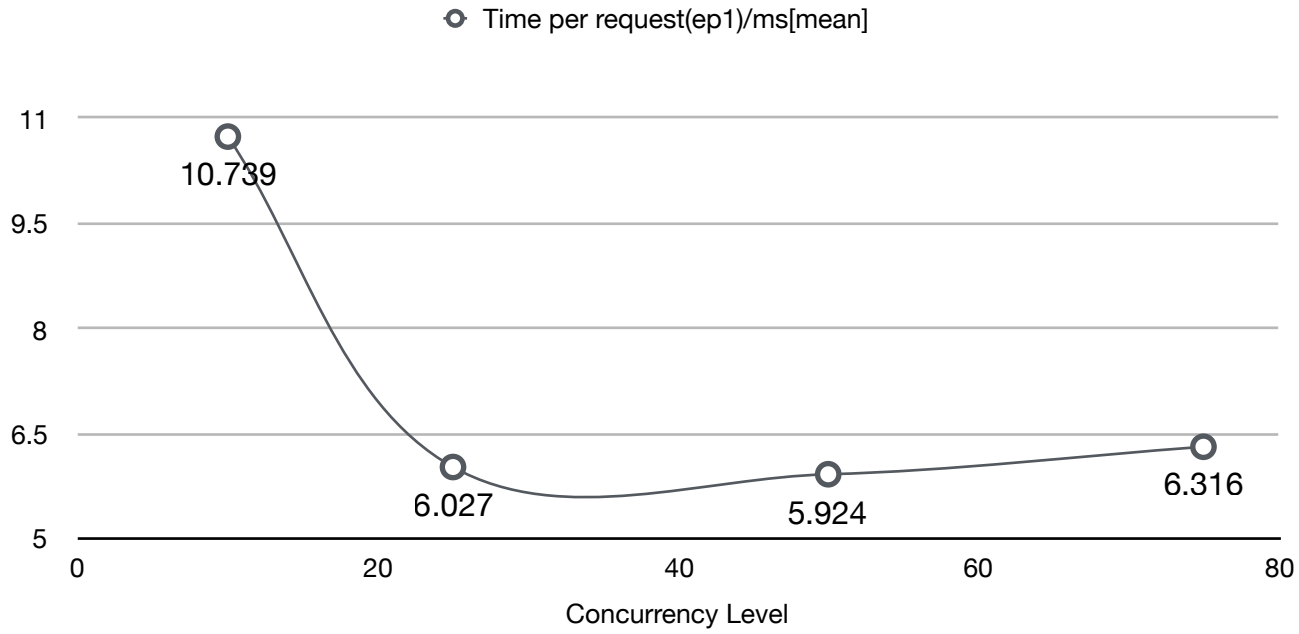
Server Software:	WSGIServer/0.2
Server Hostname:	52.14.235.109
Server Port:	8000
Document Path:	/account/login/
Document Length:	2280 bytes

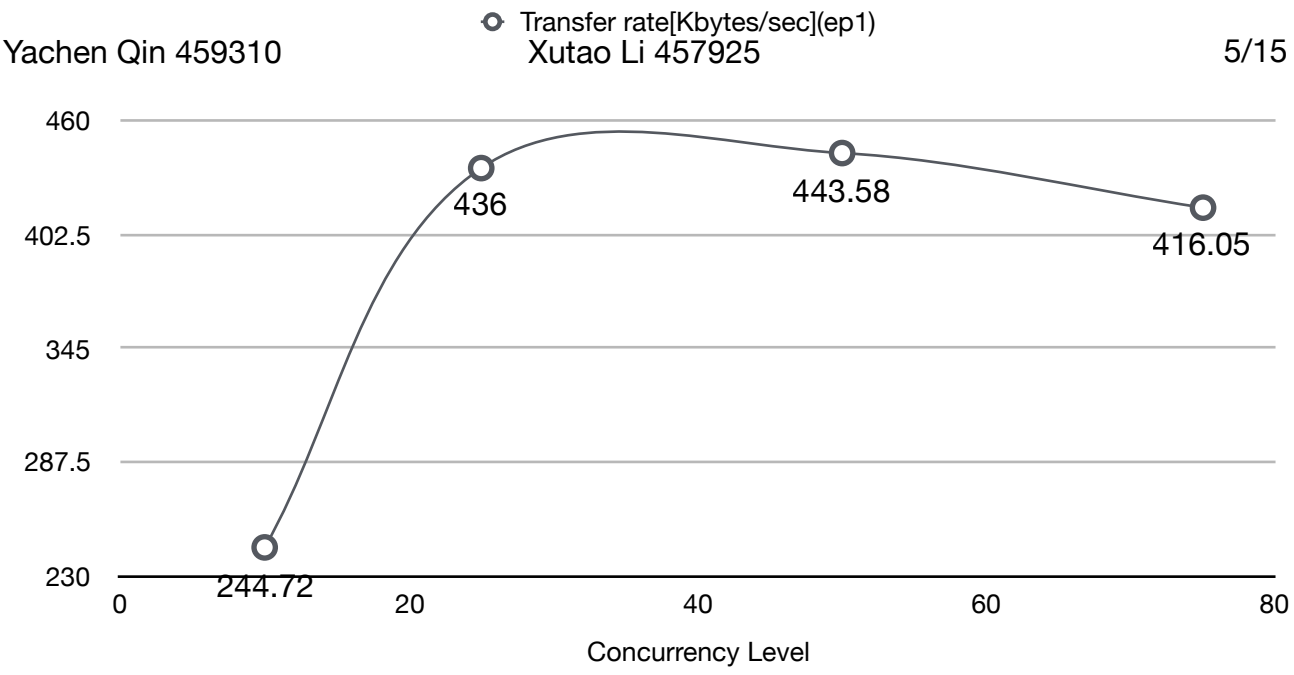


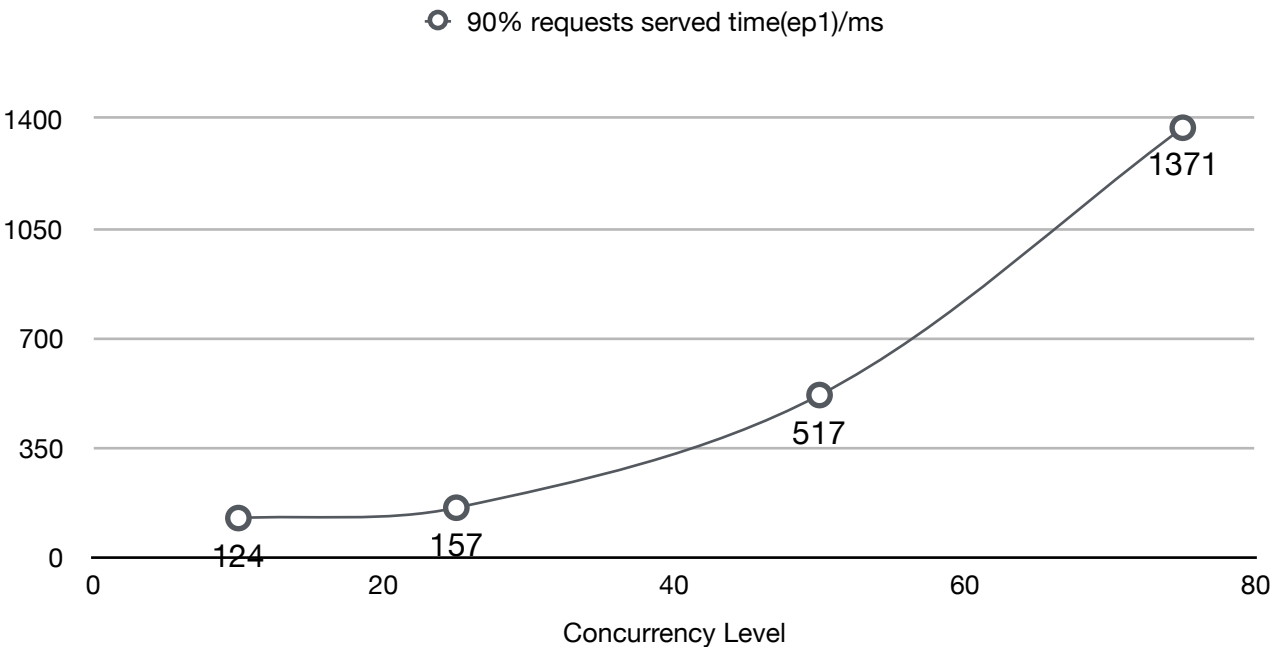



Requests per second(ep1)/sec
170
147.5
125
102.5
80
93.12
165.91
168.8
158.32
0
20
40
60
80
Concurrency Level


Time per request(ep1)/ms
500
375
250
125
0
107.386
150.683
296.217
473.728
0
20
40
60
80
Concurrency Level


Time per request(ep1)/ms[mean]
11
9.5
8
6.5
5
10.739
6.027
5.924
6.316
0
20
40
60
80
Concurrency Level





Experiment1 Result Analysis:

From the result of ab load test for our original web server we can see that the performance is not very well. The Usage of CPU is very high when the concurrency level bigger than 10 and the Requests Per Second is a little bit low. Then time per request is a little long. So we have to perform some optimization on our apache web server to see if it could become better.

Experiment2 introduction

Optimization on AWS
There are many ways to optimize our apache web-server :
- Disable or remove unused modules
- Use caching
- Use compression
- Turn off host name lookups
- Upgrade RAM and storage

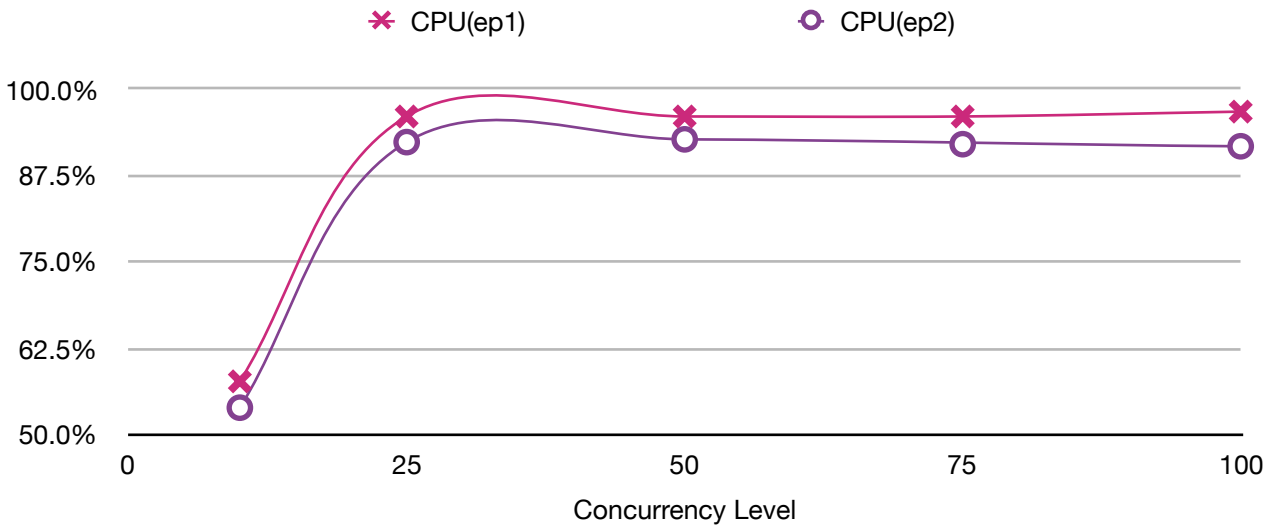
I decide to turn off the host name to see if the performance of my website become better or not. The biggest reason for Apache web server slowdowns is the time required to perform DNS lookups. Apache will record the full host name of each incoming client connection in its access.log file. Resolving each one eats up a significant chunk of time. The HostnameLookups option enables DNS lookup so that hostnames can be logged instead of the IP address.
In Apache, hostname lookups defaulted to “on,” which added latency to requests. That’s because every time a host name was encountered, a DNS lookup was required.

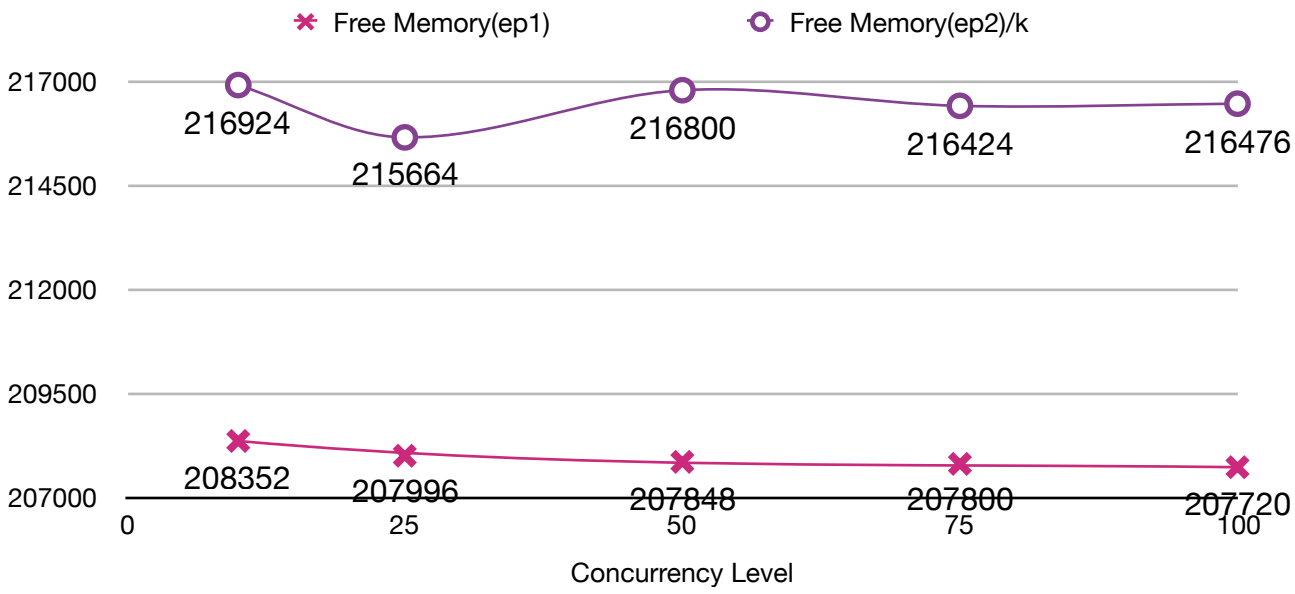
Experiment 2 Result

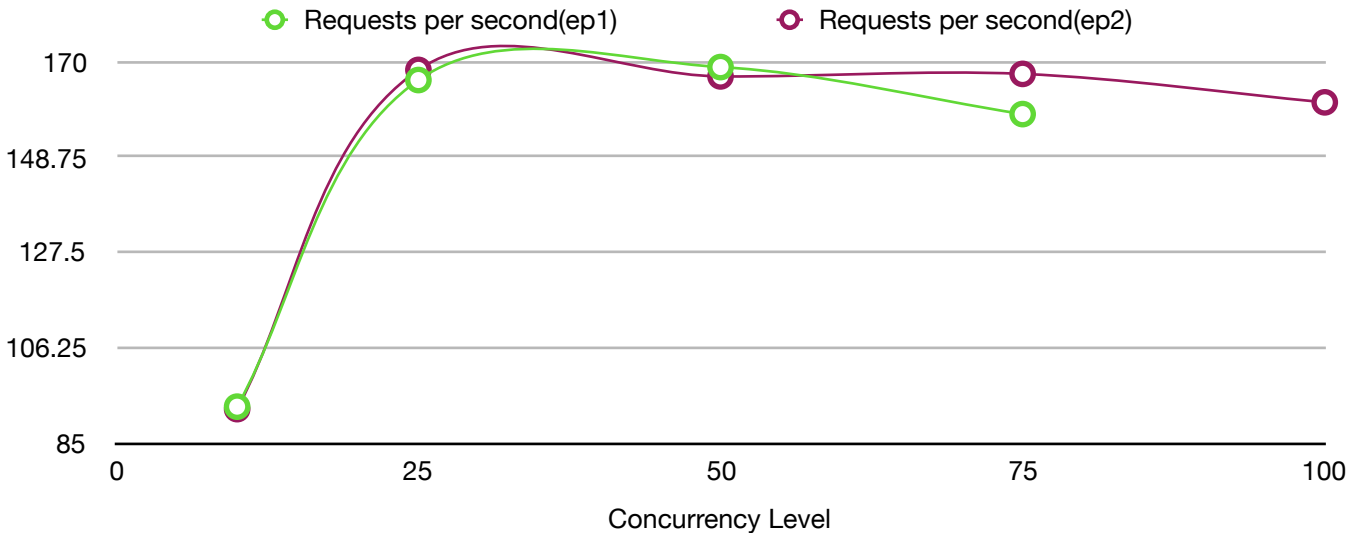
Basic Info:

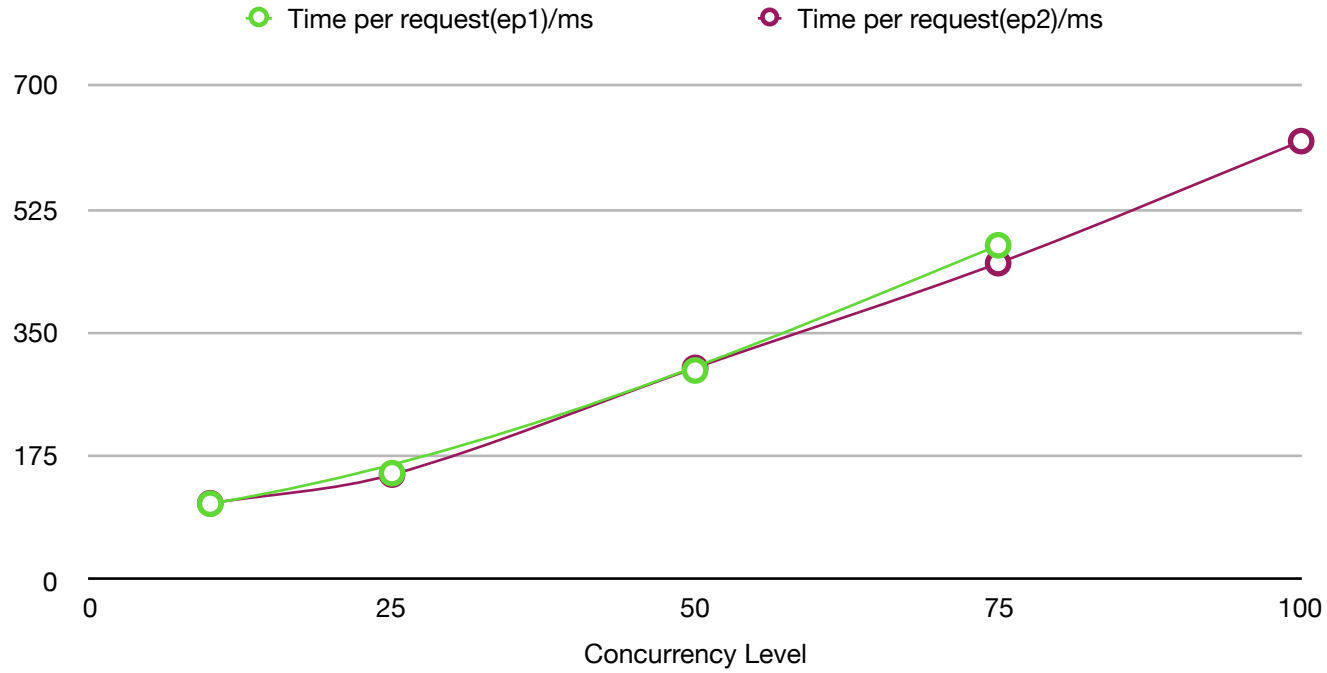
Server Software:	WSGIServer/0.2
Server Hostname:	52.14.235.109
Server Port:	8000
Document Path:	/account/login/
Document Length:	2280 bytes

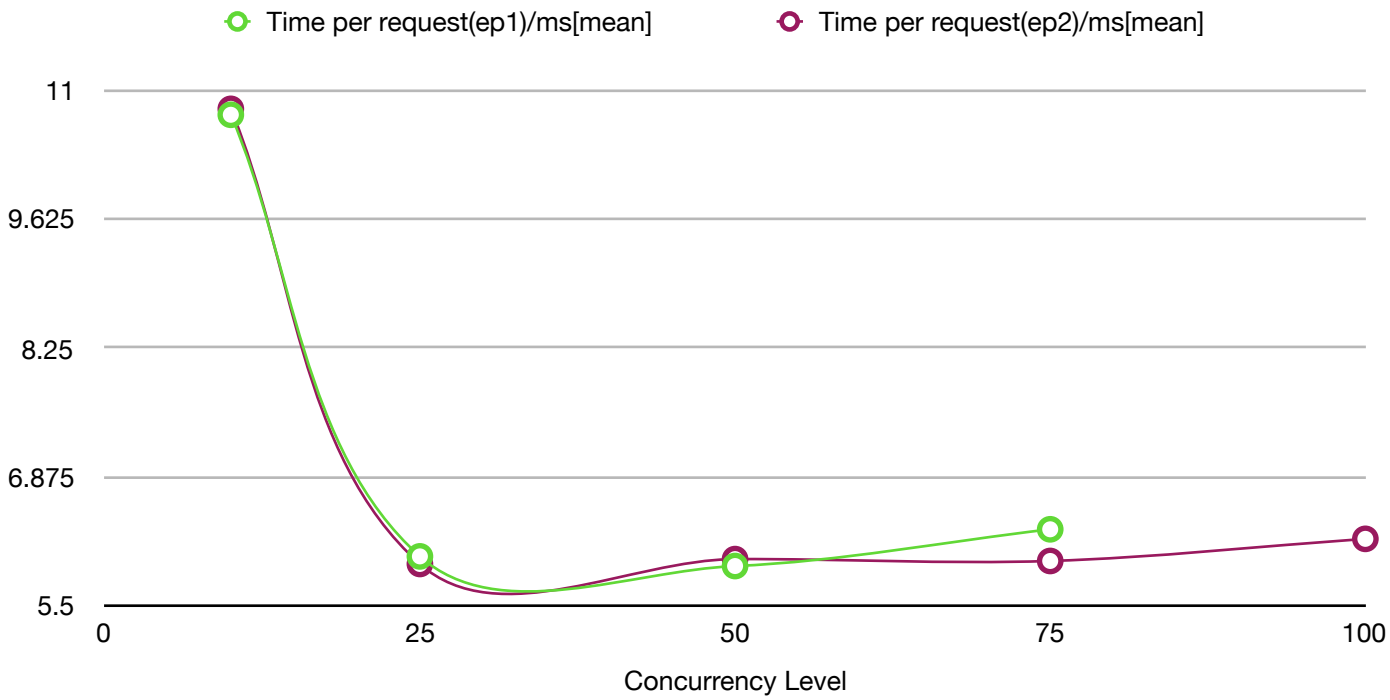
Compare between experiment1 & experiment2

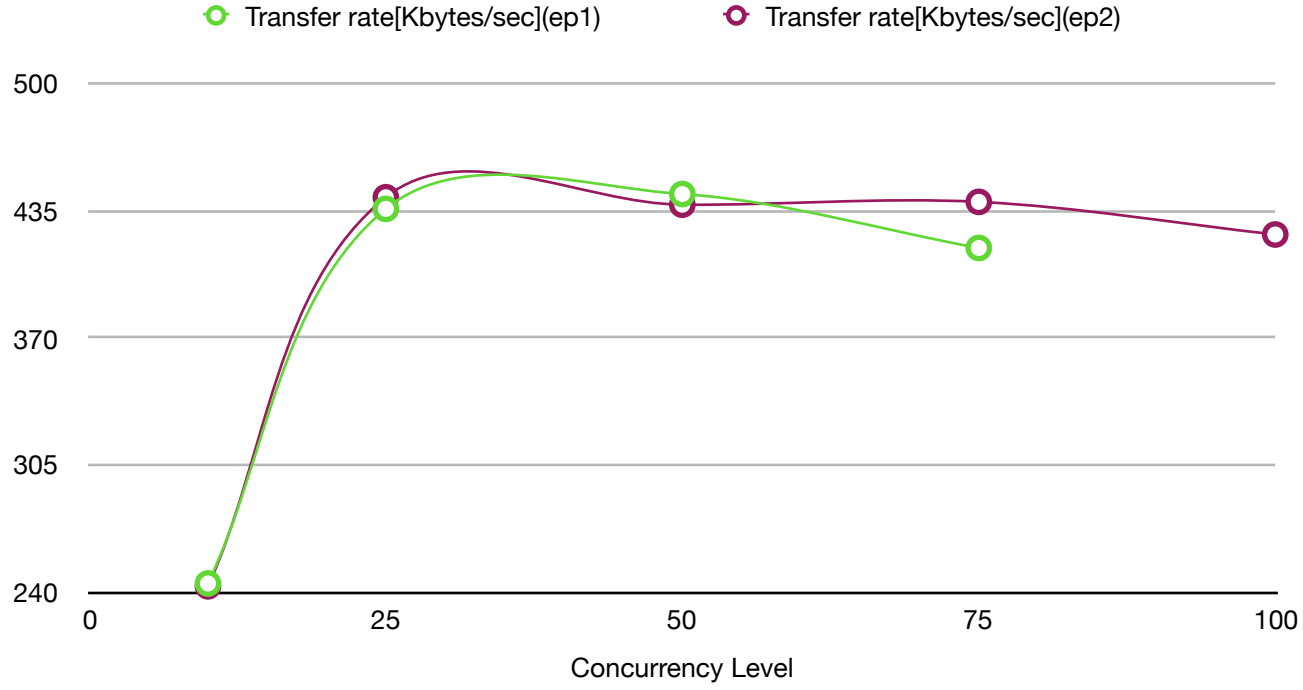


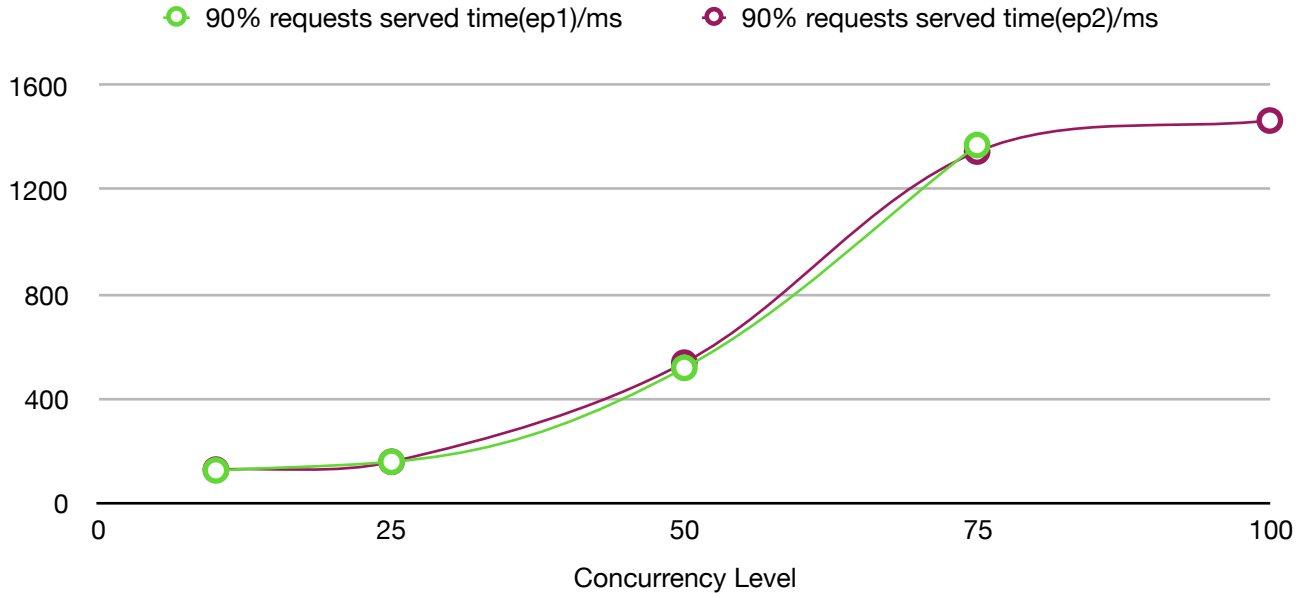

Free Memory(ep1)
Free Memory(ep2)/k
217000
214500
212000
209500
207000
216924
215664
216800
216424
216476
208352
207996
207848
207800
207720
0
25
50
75
100
Concurrency Level


Requests per second(ep1)
Requests per second(ep2)
170
148.75
127.5
106.25
85
0
25
50
75
100
Concurrency Level


Time per request(ep1)/ms
Time per request(ep2)/ms
700
525
350
175
0
0
25
50
75
100
Concurrency Level


Time per request(ep1)/ms[mean]
Time per request(ep2)/ms[mean]
11
9.625
8.25
6.875
5.5
0
25
50
75
100
Concurrency Level


Transfer rate[Kbytes/sec](ep1)
Transfer rate[Kbytes/sec](ep2)
500
435
370
305
240
0
25
50
75
100
Concurrency Level


90% requests served time(ep1)/ms
90% requests served time(ep2)/ms
1600
1200
800
400
0
0
25
50
75
100
Concurrency Level

Experiment 2 Result Analysis

From the result graph we can easily find out that the performance of web server has been approved.
The usage of CPU has decreased, with a lower usage of CPU, our web server could allowed more concurrency user at the same time. Then we can find out the free memory on our web server is also increased which means to perform the same request from the client, our server will take less memory. This could also help the web server get a higher performance.
Check the result of website performance we can see that when the concurrency level is low, below 50, the performance of Apache is similar before and after the optimization. But when the concurrency level become higher, we can see the profit of the optimization. The request per second totally increased and the time per request decreased especially the time per request for each client at one time. This will be really helpful to increase the user experience of our website.

Experiment3 introduction

Elastic Load Balancing (ELB) evaluation
In the third experiment we are going to explore the Elastic Load Balancing on our instances to see if it can help us increase the performance of web server.
Elastic Load Balancing automatically distributes incoming application traffic across multiple targets, such as Amazon EC2 instances, containers, and IP addresses. It can handle the varying load of your application traffic in a single Availability Zone or across multiple Availability Zones. Elastic Load Balancing offers three types of load balancers that all feature the high availability, automatic scaling, and robust security necessary to make your applications fault tolerant.There are three main Elastic Load Balancing tools now been used on Amazon AWS.

- Application Load Balancer

Application Load Balancer is best suited for load balancing of HTTP and HTTPS traffic and provides advanced request routing targeted at the delivery of modern application architectures, including microservices and containers. Operating at the individual request level (Layer 7), Application Load Balancer routes traffic to targets within Amazon Virtual Private Cloud (Amazon VPC) based on the content of the request.

- Network Load Balancer

Network Load Balancer is best suited for load balancing of TCP traffic where extreme performance is required. Operating at the connection level (Layer 4), Network Load Balancer routes traffic to targets within Amazon Virtual Private Cloud (Amazon VPC) and is capable of handling millions of requests per second while maintaining ultra-low latencies. Network Load Balancer is also optimized to handle sudden and volatile traffic patterns.

- Classic Load Balancer

Classic Load Balancer provides basic load balancing across multiple Amazon EC2 instances and operates at both the request level and connection level. Classic Load Balancer is intended for applications that were built within the EC2-Classic network.

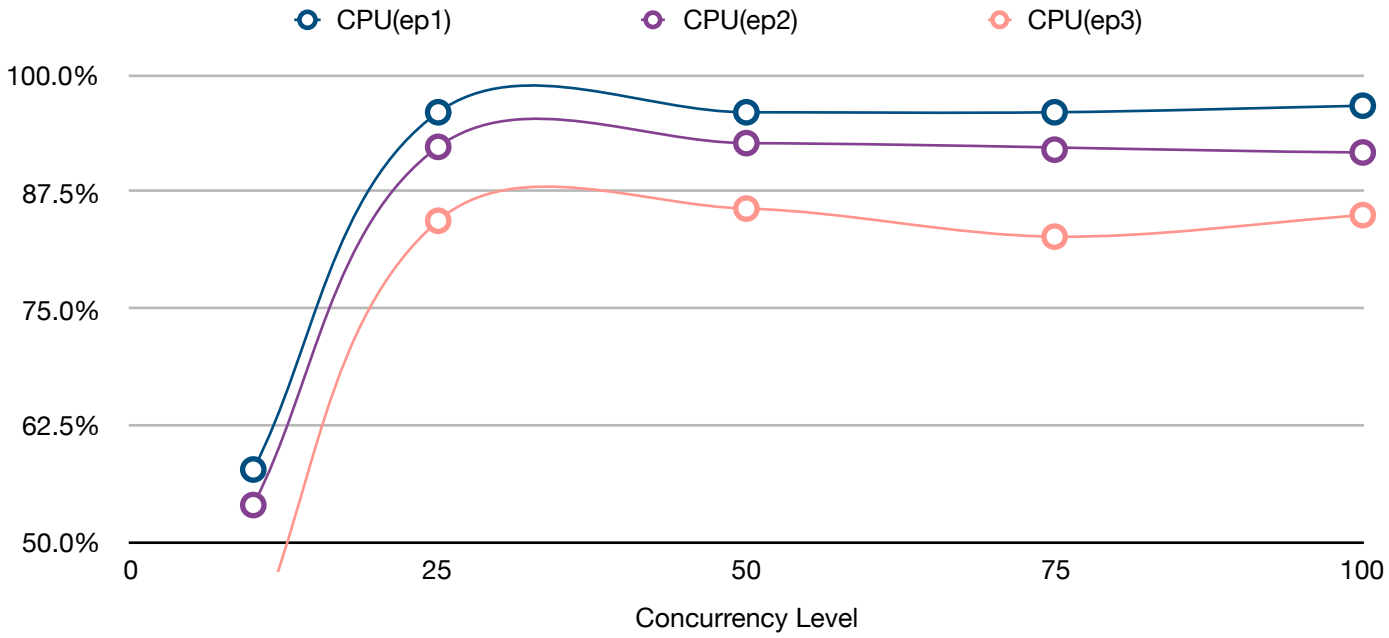
Pre-test
Since Amazon has three different types of load balancer, so we did the pre-test first to find out which balancer influence more to our website performance. From the simple pre-test result we find out that the application load balancer is helpful to influence the behavior of our web server. So we decided to use this balancer to check out how much it could do to improve our website performance.

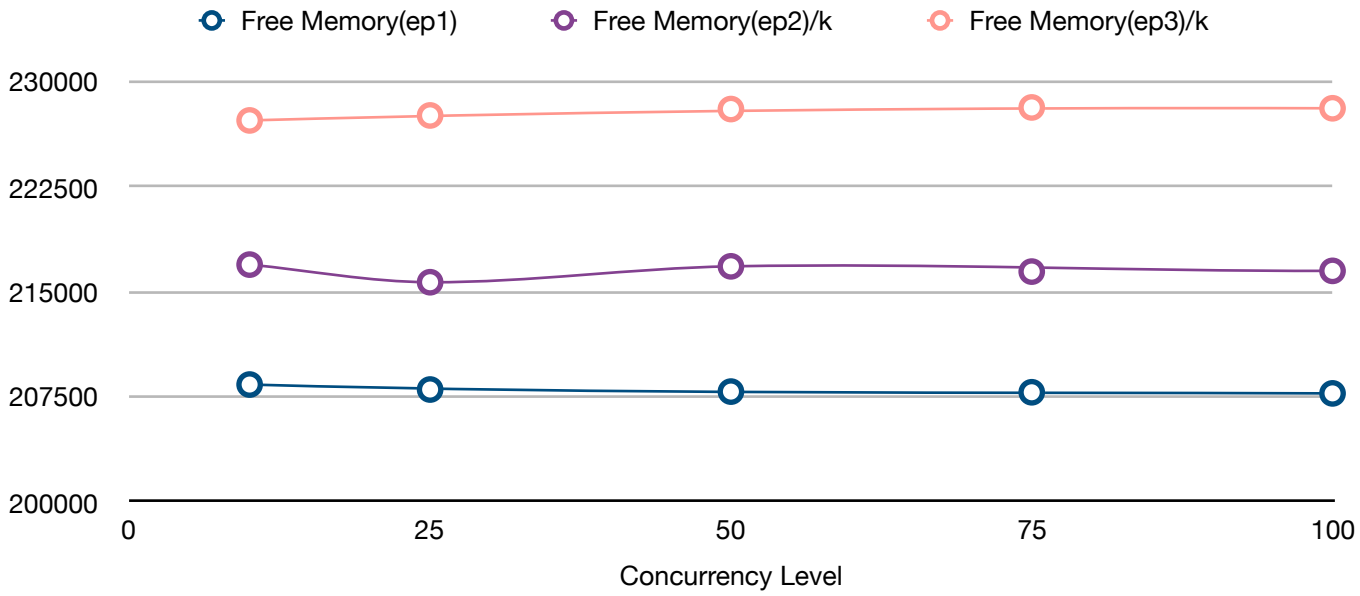
Experiment 3 Result

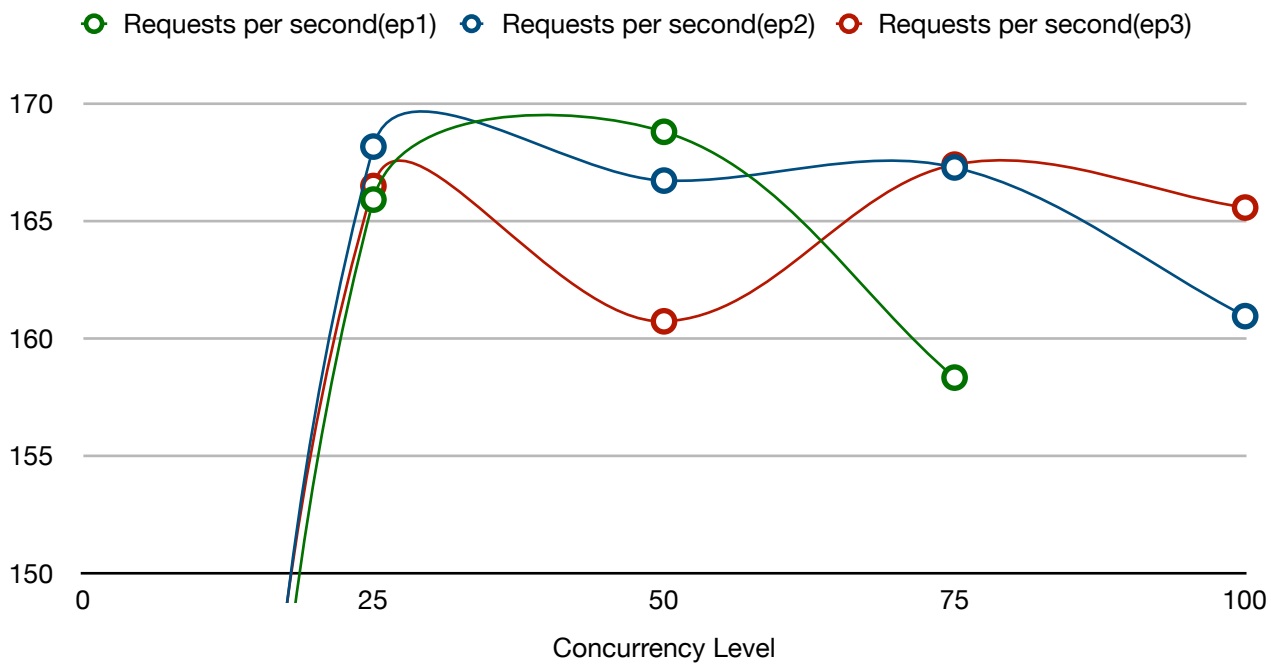
Basic Info:

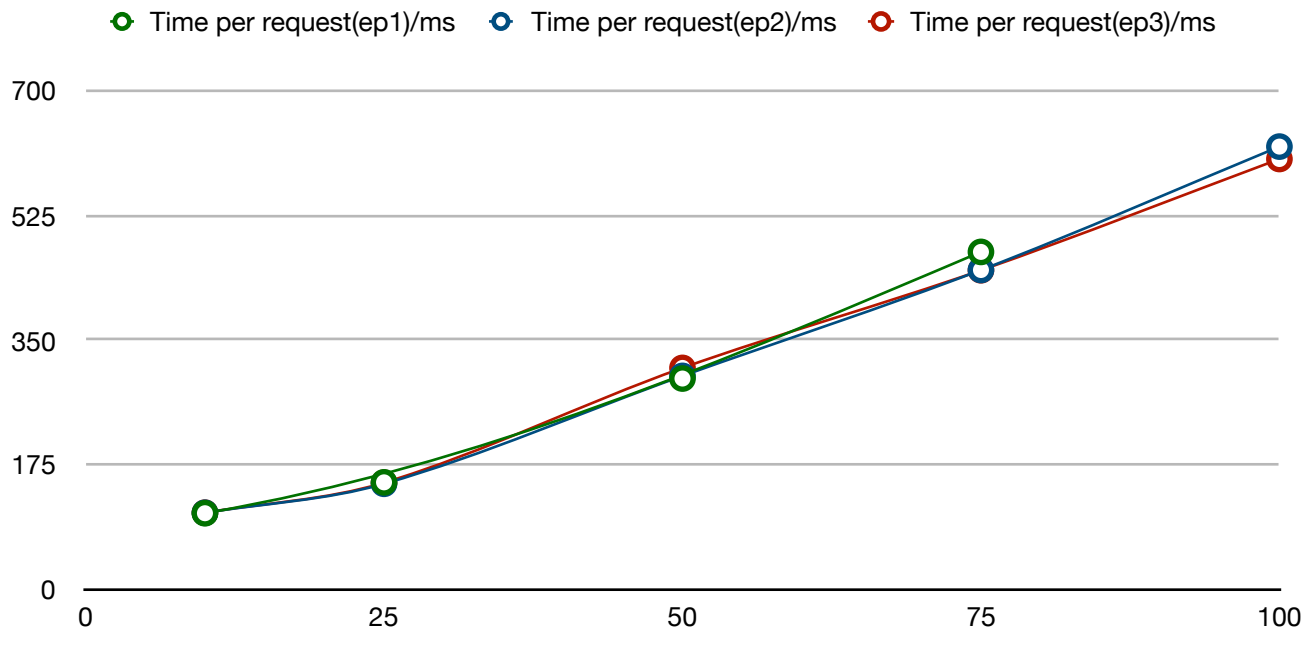
Server Software:	WSGIServer/0.2
Server Hostname:	52.14.235.109
Server Port:	8000
Document Path:	/account/login/
Document Length:	2280 bytes

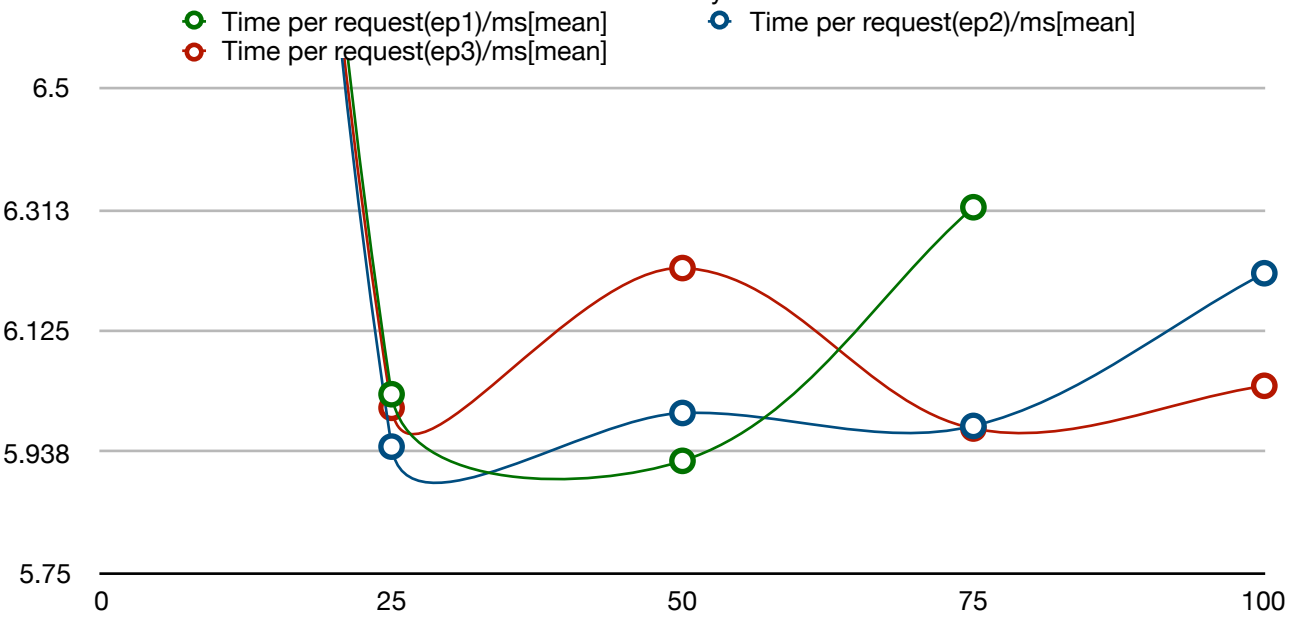
Compare between experiment1 & experiment2 & experiment

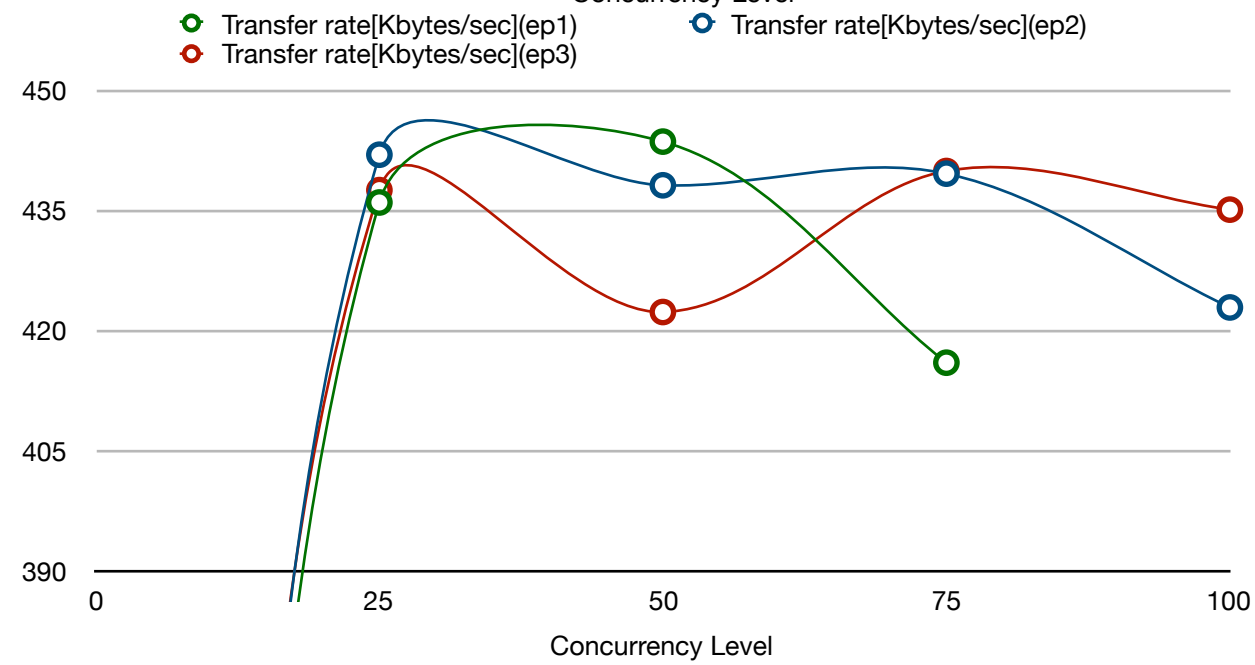


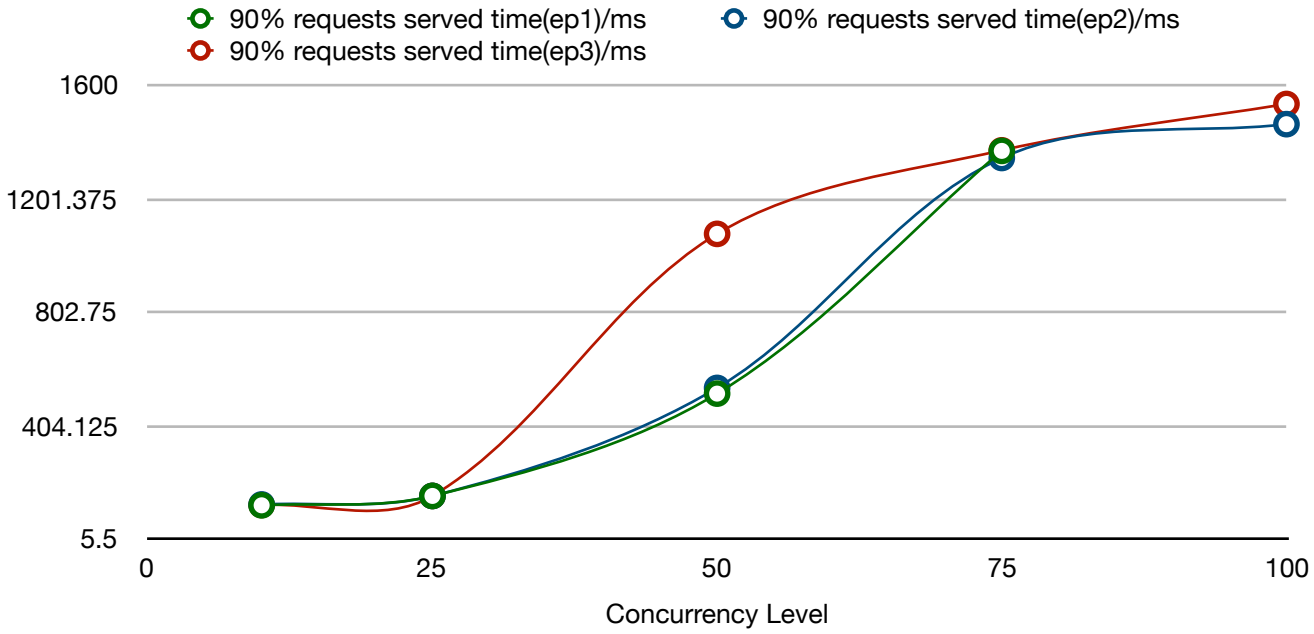





Time per request(ep1)/ms
Time per request(ep2)/ms
Time per request(ep3)/ms
700
525
350
175
0
0
25
50
75
100
Concurrency Level


Time per request(ep1)/ms[mean]
Time per request(ep2)/ms[mean]
Time per request(ep3)/ms[mean]
6.5
6.313
6.125
5.938
5.75
0
25
50
75
100
Concurrency Level


Transfer rate[Kbytes/sec](ep1)
Transfer rate[Kbytes/sec](ep2)
Transfer rate[Kbytes/sec](ep3)
450
435
420
405
390
0
25
50
75
100
Concurrency Level



Experiment 3 Result Analysis

From the result we can easily see that ELB is really helpful for our website performance.
First we can see that the usage of CPU is decrease obviously and the free memory is increasing. These results all show that with ELB, our web server will have a better performance.
From the Request per second result we can see that when the concurrency level become higher, the advantage of ELB become more obvious. Also the time for each request is also become lower. With the help of ELB, we can allow more users to get into our website in the same time without been influenced. This is really helpful to the website builder to spread his/her website to more users.

Bottleneck of our web-server

From the experiment we can see that if we don't perform any optimize operation on our web server, our website only support less than 100 users per time if the request number is 10000. But in the real world, a social website should have the capability to support more users login to the website to do different things. With the help of optimization close the DNS search, we can make the performance of our website become a little better. But in the reality, DNS is needed for the website. Or it will be a trouble for users if we only provide our IP address. Then we can use ELB to help our instance become more balancing. It could help our instances distribute different resources more efficiently. And it has a really good performance when the number of co-users increased. It can help us to get over the bottleneck of our original web server.

Apache Instances choose

An Amazon Machine Image (AMI) is a template that defines your operating environment, including the operating system. A single AMI can be used to launch one or thousands of instances.
Instances provide compute power and are the fundamental building blocks. Instances are created by launching an Amazon Machine Image (AMI) on a particular instance type. You can scale the number of instances you are running up or down on demand, either manually or automatically, using Auto Scaling.
Instance Types comprise various combinations of CPU, memory, storage, and networking capacity and give you the flexibility to choose the appropriate mix of resources for your applications. Each instance type has one or more size options that address different workload sizes. For the best experience, you should launch on instance types that are the best fit for your applications.

Instance Families are collections of instance types designed to meet a common goal. To make it easier for you to select the best option for your applications, Amazon EC2 instance types are grouped together into families based on target application profiles.

So from all those properties, the type of instances I recommend is CC2 instances. They are the latest generation of compute-optimized instances and provide the lowest cost for CPU performance for all Amazon EC2 instance types. In addition, CC2 instances provide a number of advanced capabilities: Intel Xeon E5-2670 processors; high core count (32 vCPUs); and support for cluster networking. These capabilities allowed us to create a cluster of 1064 CC2 instances that achieved a Lin-pack score of 240.09 Teraflops, good for an entry at number 42 in the November 2011 Top500 supercomputer list.

Conclusion

While it may not always yield new and shiny takeaways, load testing is a critical component to assessing the stability of new or existing applications. It is important for anyone generating tests to consider the benefits and drawbacks of different utilities to be confident about the consistency and accuracy of the results.

Appendix

The original data

Concurrency Level	CPU(ep1)	Free Memory(ep1)	Requests per second(ep1)	Time per request(ep1)/ms	Time per request(ep1)/ ms[mean]	Transfer rate[Kbytes/sec](ep1)	90% requests served time(ep1)/ ms
10	57.8%	208352	93.12	107.386	10.739	244.72	124
25	96.0%	207996	165.91	150.683	6.027	436.00	157
50	96.0%	207848	168.80	296.217	5.924	443.58	517
75	96.0%	207800	158.32	473.728	6.316	416.05	1371
100	96.7%	207720					

Concurrency Level	CPU(ep2)	Free Memory(ep2)/k	Requests per second(ep2)	Time per request(ep2)/ms	Time per request(ep2)/ ms[mean]	Transfer rate[Kbytes/sec](ep2)	90% requests served time(ep2)/ ms
10	54.0%	216924	92.57	108.022	10.802	243.28	127
25	95.3%	215664	168.17	148.655	5.946	441.95	157
50	95.7%	216800	166.72	299.899	5.998	438.14	537
75	95.0%	216424	167.28	448.357	5.978	439.59	1346
100	94.7%	216476	160.94	621.357	6.214	422.93	1465

Concurrency Level	CPU(ep3)	Free Memory(ep3)/k	Requests per second(ep3)	Time per request(ep3)/ms	Time per request(ep3)/ ms[mean]	Transfer rate[Kbytes/sec](ep3)	90% requests served time(ep3)/ ms
10	56.5%	227228	92.49	108.119	10.812	243.06	125
25	94.4%	227584	166.49	150.155	6.006	437.54	157
50	95.7%	228040	160.72	311.101	6.222	422.36	1079
75	92.7%	228156	167.40	448.024	5.974	439.92	1373
100	95.0%	228096	165.57	603.974	6.040	435.11	1536