

# **CLOUD BASED WEB RESILIENCE USING NODE.JS**

by

PRAGATI NARAYANI GUNNAM

BSCS, Andhra University, 1999

MCA, Andhra University 2002

A thesis submitted to the Graduate Faculty of the

University of Colorado Colorado Springs

in partial fulfillment of the

requirements for the degree of

Master of Science

Department of Computer Science

2017

© Copyright by Pragati Narayani Gunnam 2017

All Rights Reserved

This thesis for the Master of Science degree by

Pragati Narayani Gunnam

has been approved for the

Department of Computer Science

by

C.H. Edward Chow, Chair

Yanyan Zhuang

Sang Yoon Chang

May 15<sup>th</sup>, 2017  
Date: \_\_\_\_\_

Gunnam, Pragati Narayani (M.Sc, Computer Science)

Cloud based Web Resilience using Node.js

Thesis directed by Professor C. Edward Chow

## **ABSTRACT**

Web services are used almost everyday and for everything. Combining these with cloud based services would allow the web applications to become more available for the users. Delivering such cloud based web applications to a massive number of clients, without disruption to the service would require the application servers to run undisturbed. Such resiliency in the application servers would be desirable and at the same time a forefront consideration. We studied and experimented with Node.js web servers to achieve such resiliency. For achieving resiliency, we took into consideration of two factors: Availability and Agility. Since our experiment is in the context of cloud based web applications, we also tried locating the servers, which are geographically placed nearer to the requested client. As for our availability factor, we shut down some of the servers to prove that the client was still served with the desired response. And for our agility factor, we also calculated the duration in which the client was served with response since the request was sent. All our experiments proved that such resiliency, in fact, can be achieved using Node.js web servers.

This thesis is dedicated to the loving memory of my father, Rama Krishna Rao Gunnam.

I miss him every day, but I am glad to know he always believed in me which gets me through the hard times.

# ACKNOWLEDGEMENTS

First and foremost, I would like to sincerely thank my advisor Dr. Edward Chow for all the guidance and interest he took in the progress of this work. I am very grateful to him for his constructive comments, feedback and for always being there to help either in person or online. I would like to thank him also for providing the resources like Amazon Web Services and helping me set them up during our research.

I am extremely grateful to Dr. Yanyan Zhuang and Dr. Sang Yoon Chang for their valuable suggestions and advises in our thesis proposal and during all of this thesis work without which it would have been very difficult for me to come up with this great work.

I would like to extend my sincere thanks to my mentor, Mr. Weston Pace for introducing me to Node.js. My deep appreciation goes to him for his patience in guiding me and helping me understand Node.js, which further helped me progress in my thesis.

Many thanks go to my family members for their constant support and encouragement. I am grateful to my parents for all their love, affection and blessings without which I would not have gotten this far in life. And finally a special note of thanks goes to Prasad, my husband and my kids, Pranav and Praful for their continual encouragement, support, advice and patience that has enabled me to accomplish things I never thought were possible. Thank You.

# TABLE OF CONTENTS

## CHAPTER

I. INTRODUCTION .....	1
1.1    Goal of Thesis .....	2
1.2    Node.js .....	3
II. BACKGROUND.....	4
III. DESIGN .....	6
3.1    Request.....	8
3.2    Promise .....	10
IV. IMPLEMENTATION.....	12
V. IMPROVEMENT & PERFORMANCE EVALUATION.....	15
5.1    Design Improvement.....	15
5.2    Performance Evaluation.....	16
VI. LESSONS LEARNED & CHALLENGES FACED .....	29
6.1    Better Understanding of Node and packages.....	29
6.2    Better Understanding of Promise.....	30
6.3    Better understanding of Amazon EC2 Instances .....	31
VII. FUTURE DIRECTIONS.....	32

VIII. CONCLUSION .....	33
REFERENCES .....	35
APPENDIX A.....	37
APPENDIX B .....	41



# TABLE OF FIGURES

## FIGURE

<b>1:</b> A Simple Overview of our design.....	7
<b>2:</b> A Detailed Description of the Design .....	14
<b>3:</b> Performance from both the original and improved design for a client in Colorado Springs to a Server in EAS private cloud running on port 8000 .....	18
<b>4:</b> Performance from both the original and improved design for a client in Colorado Springs to a Server in EAS private cloud running on port 9090. ....	18
<b>5:</b> Performance from both the original and improved design for a client in Colorado Springs to a Server(Amazon Instance) in Oregon (West) region running on port 9000.....	19
<b>6:</b> Performance from both the original and improved design for a client in Colorado Springs to a Server (Amazon Instance) in Ohio (East) region running on port 8080.....	19
<b>7:</b> Performance from both the original and improved design for a client in Singapore to a Server(Amazon Instance) in N.Virginia (East) region running on port 8000.....	20
<b>8:</b> Performance from both the original and improved design for a client in Singapore to a Server(Amazon Instance) in N.Virginia (East) region running .....	20
<b>9:</b> Performance from both the original and improved design for a client in Singapore to a Server (Amazon Instance) in Oregon (West) region running on .....	21
<b>10:</b> Performance from both the original and improved design for a client in Singapore to a Server (Amazon Instance) in Ohio (East) region running on port 8080. ....	21
<b>11:</b> Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in N.Virginia (East) region running on port 8000. ....	22
<b>12:</b> Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in N.Virginia (East) region running on port 9090. ....	22
<b>13:</b> Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in Oregon (West) region running on port 9000. ....	23

<b>14:</b> Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in Ohio (East) region running on port 8080. ....	23
<b>15:</b> Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in N.Virginia (East) region running on port 8000 .....	24
<b>16:</b> Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in N.Virginia (East) region running on port 9090 .....	24
<b>17:</b> Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in Oregon (West) region running on port 9000. ....	25
<b>18:</b> Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in Ohio (East) region running on port 8080. ....	25
<b>19:</b> Differences in total time for clients to Oregon server, during different times of the day. ....	27
<b>20:</b> Differences in total time for clients to Ohio server, during different times of the day. ....	28
<b>21:</b> Differences in total time for clients to Virginia server, during different times of the day. ....	28

# TABLE OF NPM PACKAGES

## NPM PACKAGE

1: Using 'request' .....	9
2: Using 'bluebird' Promise .....	11

# CHAPTER 1

## INTRODUCTION

Web services became so important in our daily life. The introduction of cloud-based web servers reduces the cost of maintaining such web services. By choosing the hosting regions/data centers, we improve the performance of response time of web services. By using their new load balancing features to form a web-cluster, we improve the availability and resilience of the web services. To improve the performance and maintenance, migration of Virtual Machines (VMs) were proposed, to move VM to a new host or hosts in a new data center within a region. Some even proposed to perform live virtual machine migration with the goal of no disruption in service during the migration (Clark, 2005).

Virtual Machines offer many services which made data storage very easy, efficient and less expensive. Many cloud providers offer virtual machine hosting, cloud data and database services for prices which are imaginatively less than what we had before. These services made cloud a popular choice for many small and large organizations and even for individuals to store their personal data. The cloud is made up of VMs in two different flavors: VMs which are located on a single physical platform and

VMs which are located in different geographical places. Migration of VMs is done when there is a need for optimizing: CPU utilization, storage, power consumption, server and network resources to optimize application performance and also bandwidth utilization efficiency. Migration of VMs usually deals with transferring the instance image and its data from VM (source VM) to another VM (destination VM).

## **1.1 Goal of Thesis**

Live migration is the process in which the transition of a VM from source to destination is done without halting the guest operating system [Clark2005] (Clark, 2005). This live migration technique is usually done by copying memory pages (kernel internal state and application level state) of the source VM while it continues to run. When the hypervisor of the source VM decides that this transfer is complete, it will halt the VM, and trigger the requests to be sent over to the destination VM. Now the destination VM starts executing and all the 'dirty' data would be updated. There might be a possibility for these source and destination VMs to be present at the same physical server or at a different geographical location.

Instead of touching at kernel level, we focus on providing web service resilience at the application level. By focusing on the application level, we allow our design to be portable across different cloud providers and VM platforms.

We choose Node.js [Tilkov2010] (Tilkov, 2010) for developing the load balancer, the resilient module and the web servers. It has the advantage of using same programming language (JavaScript) for portability and potential migration of node.js modules across different hosting platforms, possibly even running them at client

machines. Node.js core functionalities are kept to a minimum and all the existing APIs expose minimum amount of complexity to the program. For complexity tasks, you can pick, install and use several third-party modules [Teix2012] (Teixeira, 2012.) or develop the JavaScript code yourself.

## 1.2 Node.js

Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world. As an asynchronous event driven JavaScript runtime, Node is designed to build scalable network applications.

Node.js brings event-driven programming to web servers, enabling development of fast web servers in JavaScript. Developers can create highly scalable servers without using threading, by using a simplified model of event-driven programming that uses callbacks to signal the completion of a task. Node.js was created because concurrency is difficult in many server-side programming languages, and often leads to poor performance.

We chose Node.js for our experiment because of its scalability, agility and of its close association with JavaScript[Tilkov2010] (Tilkov, 2010).

# CHAPTER II

## BACKGROUND

During our study about live migration of virtual machines, we came across several authors who studied and researched about its issues, shortcomings, solutions and various insights. In 2005, Clark et al [Clark2005] (Clark, 2005), researched about live migration and found out that the downtime for each live migration of the operation system was very less. They even proposed of recommending to perform live migration of operation systems running interactive loads.

In 2009, Voorsluys et al [Voorsluys2009] (Voorsluys, 2009), researched the effect of live migration on the virtual machines hosting web servers. They evaluated the effects of live migration of virtual machines on the performance of applications running inside Xen VMs. Their results show that, in most cases, migration overhead is acceptable but cannot be disregarded, especially in systems where availability and responsiveness are governed by strict Service Level Agreements [Voorsluys2009] (Voorsluys, 2009). Later on, much studies are done focusing primarily on the security issues that might arise during live migration. In 2014, Navamani et al [Nava2014] (B. Navamani, 2014),

experimented with the resources involved in live migration and found out the security does not hold true once the live migration starts.

The ultimate aim of all the above studies was to achieve resiliency for providing an uninterrupted service to the client. But since all of the above studies involve system level migration to achieve resiliency, we choose to research on the application level. So, instead of touching at kernel level, we focus on providing web service resilience at the application level. By focusing on the application level, we allow our design to be portable across different cloud providers and VM platforms. We chose Node.js for our experiment, because of its asynchronous I/O model, and JavaScript, since it supports callback functions[Tilkov2010] (Tilkov, 2010). Using cloud based approach for serving web applications, we focus on achieving resiliency for such applications.



# CHAPTER III

## DESIGN

In order to achieve web resiliency, we suggest the following design for our intended experiment. Our design consists of multiple redundant Node.js servers located in different geographical regions. For our experiment, we would be setting up:

- three Node.js servers within our EAS VI private cloud and
- each Node.js server on two Amazon EC2 Instances in East region within AWS Cloud and
- one Node.js server on one Amazon EC2 Instance in West region within AWS Cloud

Within Amazon EC2 Ohio region, we would like to have one instance to act as a low-level load balancer for the other instance created within that region. Any requests coming to that low level load balancer should be directed to the other instance for the response and that response should be served back to this low level balancer before serving back to the high-level load balancer.

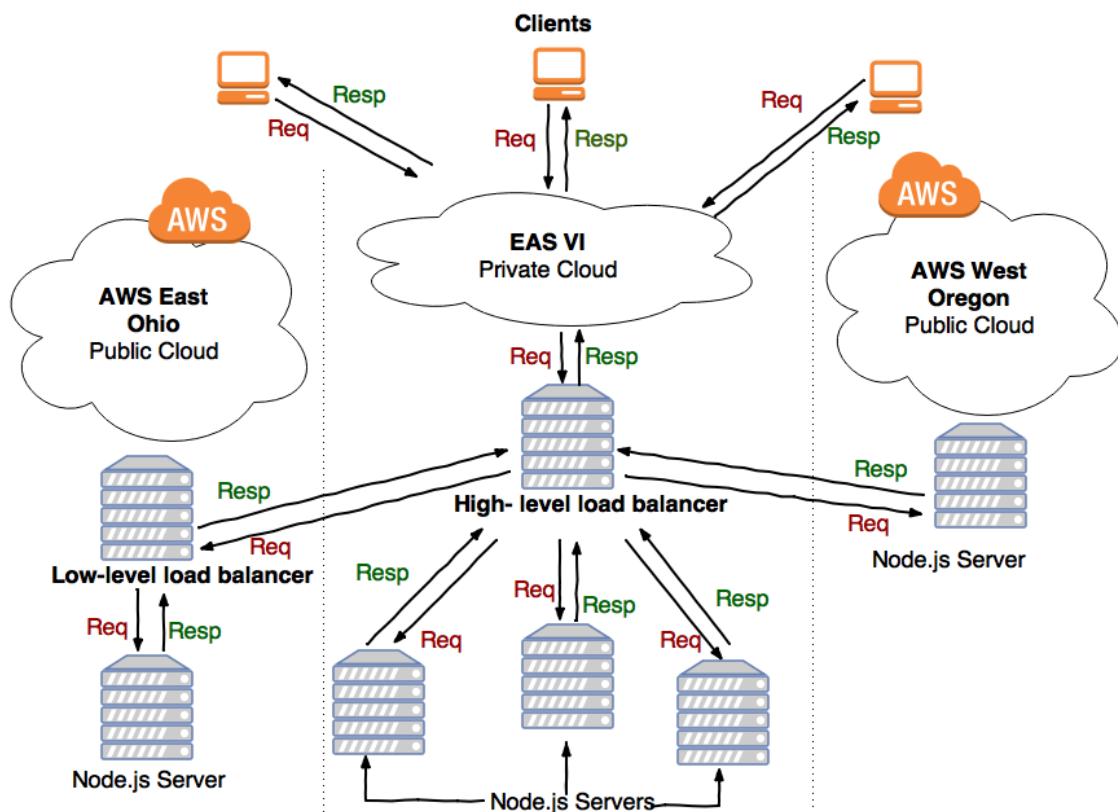
One of the web servers within our EAS VI private cloud would act as the high-level load balancer, which would keep track of the remaining servers on EAS private cloud along with the low level load balancer (Amazon Instance) in Ohio and the other Amazon instance in Oregon.

This high-level load balancer would keep track of the servers of their availability, in the sense, that if any server is unresponsive, the load balancer should be able to bring up that server.

It should also have the ability to shut off a server, if need be. In addition to this, it should also be able to secure connect to the Amazon Instances.

We desire to achieve the maximum resiliency using this high-level load balancer. For this, we would initially calculate the distances of the servers from the client's IP address. In the ascending order of the distances, we list our servers accordingly. Secondly, we would send the request to all of these servers in that order. All the responses from all the servers should reach this load balancer and it should decide which one of the responses should be sent back to the client, especially in the context of shortest duration (Please Refer to Figure1).

Our simple overall design would look as follows:



*Figure 1: A Simple Overview of our design*

When a user sends a request to the server (here it is EAS VI private cloud), this request is intended to travel in the following way in order to have the response back:

- Initially the request would be handled by the high-level load balancer by sending it to all of the available Node.js servers.
- These available Node.js servers would be approached in the ascending order of their geographical distances from the requested server (the user's server).
- Each server would serve the request.
- If any of the available server is serving as the low-level load balancer, the request would be sent to its servers and wait for the response to be sent back to it.
- Calculate the time taken for the request/response.
- Whichever server serves the request in the shortest time, the response from that server would be sent as response to the client.

In order to achieve web resiliency, the high-level load balancer should have the following abilities:

- Should know the status of the servers – responsive/unresponsive.
- Should have the ability to bring up a server.
- Should have the ability to shut off a server.
- Should be able to secure connect to a remote server (Amazon instances)

To achieve the desired flow of the request and to receive the response better, we would be using the package '*request*' in node and also '*Promise*' as an alternate to callbacks.

### 3.1 Request

Request is used to make http calls easier and simpler. It also supports HTTPS and make redirects by default. It can be installed within node using: `npm install request`

This ‘request’ should be imported into our application and be used as:

```
var request = require('request');

request (options, callback)
// where options= {url: 'http://google.com'}

OR

request('http://www.google.com', function (error, response,
body) {
    if (error) {
        console.log ('error:', error);
    } else {
        console.log('elapsedTime:', response.elapsedTime);
        console.log ('body:', body);
    }
});
```

### ***NPM Package 1: Using 'request'***

This request takes in an ***url*** or an ***options*** object and a ***callback*** function as parameters.

The ***options*** object should contain a ‘***url***’ property. If the request to the ***url*** is a success, response is returned as an object ‘***response***’ and the data of the response as ‘***body***’. This ***response*** object has many properties and some of them which we might find use of:

- **.statusCode** – status code of response if response was received
- **.elapsedTime** – if time is set to true in ***options***, it adds a property ***elapsedTime*** to ***response*** object.
- **.body** - entity body for request.

We use:

- ***statusCode*** to find the availability of a server. If the ***statusCode*** is 2xx, it is a successful request. If the code is 4xx, it is a client error and if the code is 5xx, it is a server error.

- *elapsedTime* is the duration of the entire request/response in milliseconds
- *body* is the response's data.

## 3.2 Promise

Promises are basically used as an alternative to callbacks in JavaScript. A promise represents the result of an asynchronous operation. A promise is in one of three different states:

- *pending* - The initial state of a promise.
- *fulfilled* - The state of a promise representing a successful operation.
- *rejected* - The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

We installed package 'bluebird', especially for using its advanced features associated with Promises: `npm install bluebird`

By returning a *Promise*, we now have access to a *value* representing the asynchronous operation (the promise). We can pass the promise around and anyone with access to the promise can consume it using ***then*** *regardless if the asynchronous operation has completed or not*. We also have guarantees that the result of the asynchronous operation won't change somehow, as the promise will only be resolved once (either fulfilled or rejected). Think of ***then*** as a function that *unwraps* the promise to reveal what happened from the asynchronous operation. Anyone with access to the promise can use ***then*** to unwrap it. In addition, Promise provides us with ***catch***, which can be used for error handling. We discuss below how to use promise as an alternative to callbacks:

### Using Callback:

```
request('http://www.google.com', function (error, response, body) {
  if(error) {
    console.log('error:', error);
  } else {
    console.log('body:', body);    // Should use the values here
  }
});
```

### Using Promise:

```
var Promise = require('bluebird')

function getRequest(){
  return new Promise(function(resolve, reject) {
    request('http://www.google.com', function (error, response, body) {
      if(error) {
        reject(error);
      } else {
        resolve(response); // Can use the value of promise later
      }
    });
  });
}

getRequest()
  .then( response => {          // Using the value of promise here
    console.log('body:', response.body);
  }).catch( error=> {
    console.log('error:', error);
  });
```

### *NPM Package 2: Using 'bluebird' Promise*

# CHAPTER IV

## IMPLEMENTATION

For our implementation of our design, we make use of both *request* and *promise*.

Initially, when the high-level load balancer receives a request from a client (1),

- a. It calculates the geographical distances from the client's IP address to each available server's IP addresses.
- b. Sorts the servers in ascending order of these distances.
- c. Now sends the request to each server (2).
- d. If the server can serve the incoming request, a Promise object with *elapsedTime* and *body* will be sent to the high-level load balancer, otherwise an error object would be sent.
- e. Once it receives all the Promise objects (some of them might be an error object if the server fails to serve), it will try to resolve/fulfill these promises simultaneously using *Promise.all ()* functionality. This functionality would resolve all the promises using *then* and return one promise object which would be an array containing all the resolved values. (We have to note that using

Promise.all ([Promise Objects]) would resolve only when all the promises are fulfilled; even with one rejected promise, the whole result would be rejected. So we modified our functionality in a way that when a request is rejected, we converted it into a Promise object and tried to resolve (whose end result would be an error message pushed into the final Promise array).

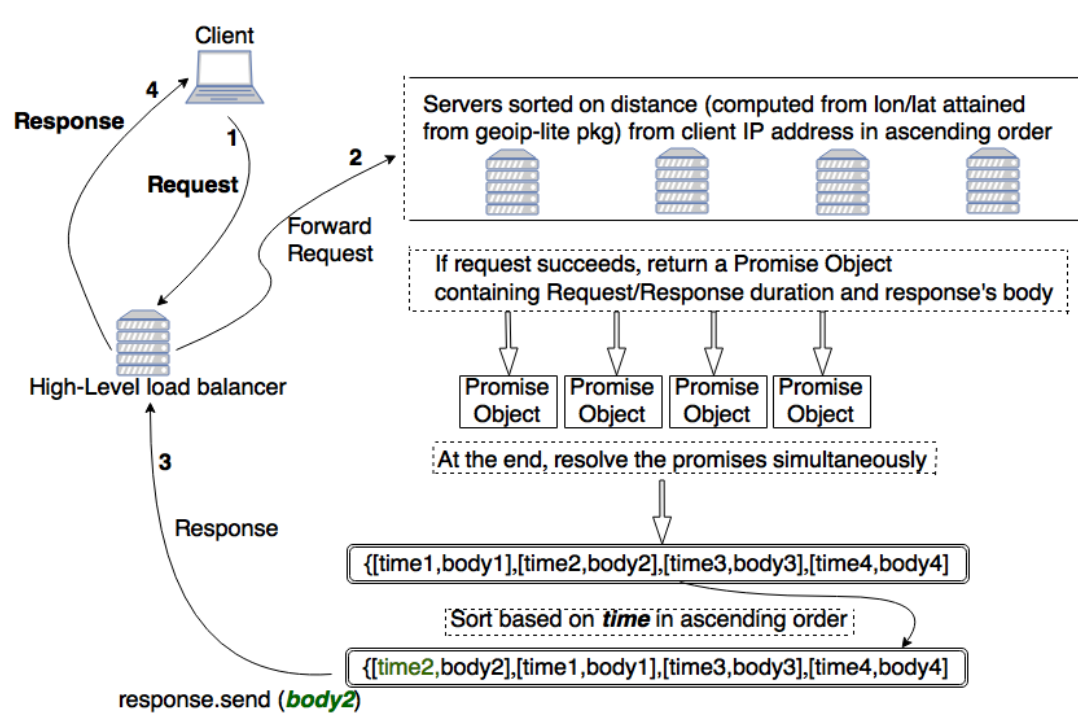
- f. We then removed any value in the final Promise array which starts with string 'Error:', so that the final Promise array would now hold only the fulfilled values.
- g. We then sort the array based on the *elapsedTime* value column.
- h. The *body* associated with the first *elapsedTime* value in the now sorted array would be sent to the high-level load balancer as a final response (3).
- i. This response would be sent to the client (4).

This would make sure that the client always receives the response from the server that has the shortest request/response duration. We intentionally shut down some of the servers to find out the result. Even if one of the server does not respond, this design makes sure the client is served indefinitely (Please Refer to Figure 2).

We performed our experiments by sending a request for test files with sizes ranging from <1 Kb to <= 54 Mb. No matter the size, the client was always served with the response. This would prove the ***availability*** of the service for the clients in a satisfactory way.

But while considering the factor of resiliency, this design does not necessarily serve the client in the fastest way possible. We had to modify our design for optimum resiliency factor: ***agility/speed***.





*Figure 2: A Detailed Description of the Design*

# CHAPTER V

## IMPROVEMENT & PERFORMANCE EVALUATION

### 5.1 Design Improvement

Our design would serve the client with the response only after finding out the durations of the request/response from each available server. We have to observe, even though the response from a given server, which is geographically nearer to the client, would send the response in a shortest time than other servers, that response would be sent to the client only after all the durations from all the other servers are calculated. This would delay in serving the response to the client.

So, to improve the design, we sent out the response to the client immediately as soon as the request sent to the first server, which is geographically nearer to the client, was resolved. If in any case, the server is down or the service is disrupted, the response from the next available server would be sent to the client and so on. This improvement in design would serve the client within shortest possible time.

## 5.2 Performance Evaluation

### 5.2.1 Improved Design

Now for the next step, to evaluate this improved design in terms of better performance and availability, we calculated the request/response durations from clients, who are located in four different geographical locations, to the servers. We selected requesting clients from:

1. Colorado Springs
2. Singapore
3. North California
4. Tokyo

For the clients other than from Colorado Springs, since they couldn't access our EAS VI private cloud's high-level load balancer, we set up an Amazon EC2 Instance in North Virginia region to act as a high-level load balancer for the clients from Singapore, North California and Tokyo. This high-level load balancer (running on port 8080) would be replacing the functionality we currently have via EAS VI private cloud. Since we couldn't access the private cloud for our experiment from outside of it, we recreated the functionality of high-level load balancer and web servers within the North Virginia Amazon Instance. This load balancer would be monitoring two other web servers under it, running on ports 8000 and 9090 respectively.

We sent requests from all the clients to both our original design and the improved design and evaluated their performances. We collected the data by sending the request to

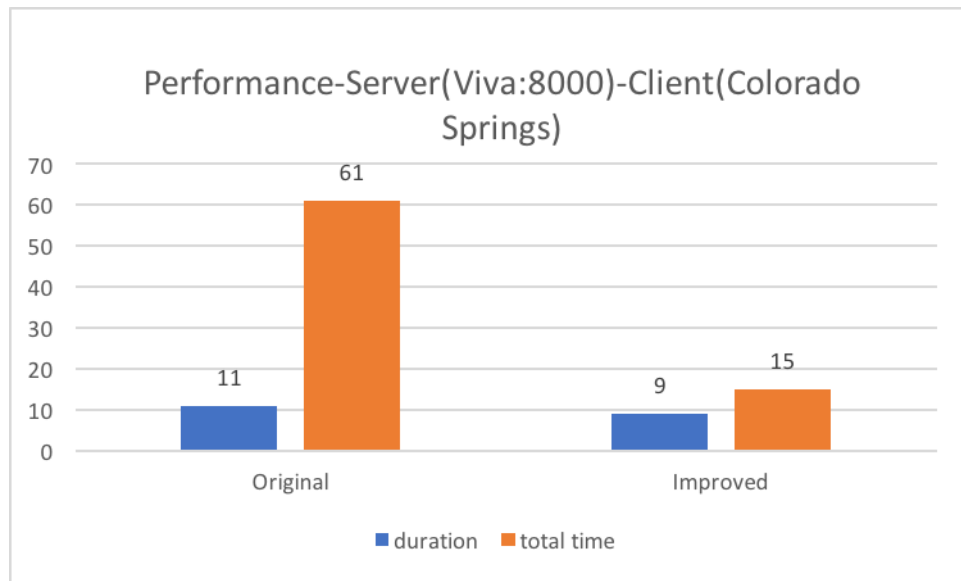
the high-level load balancer – either on EAS VI private cloud or on North Virginia Amazon EC2 Instance, Amazon EC2 Instance in Oregon region and Amazon EC2 Instance in Ohio region. We sent the requests from the client to our original design, which would send the response after all the available webserver's duration of request/response would be calculated and then that response would be sent to the client. Also we sent the same requests to our improved design which would send the response as soon as the request is sent to the first available server.

We calculated:

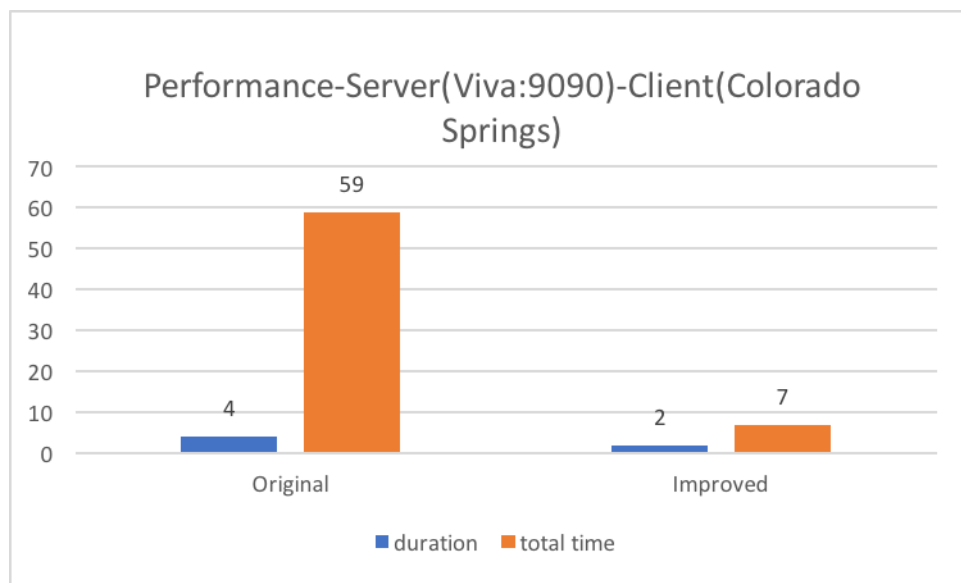
- the *duration*: the total time it took the *server* to *get* the response since it received the request, in milliseconds (duration of request/response). And,
- the *total time*: it took for the client to receive the response since the request was sent from the client, in milliseconds.

The *total time* for the client to receive the response in both the original and improved design are highly differentiable. Our improved design reduced the *total time* drastically for the client to receive the response.

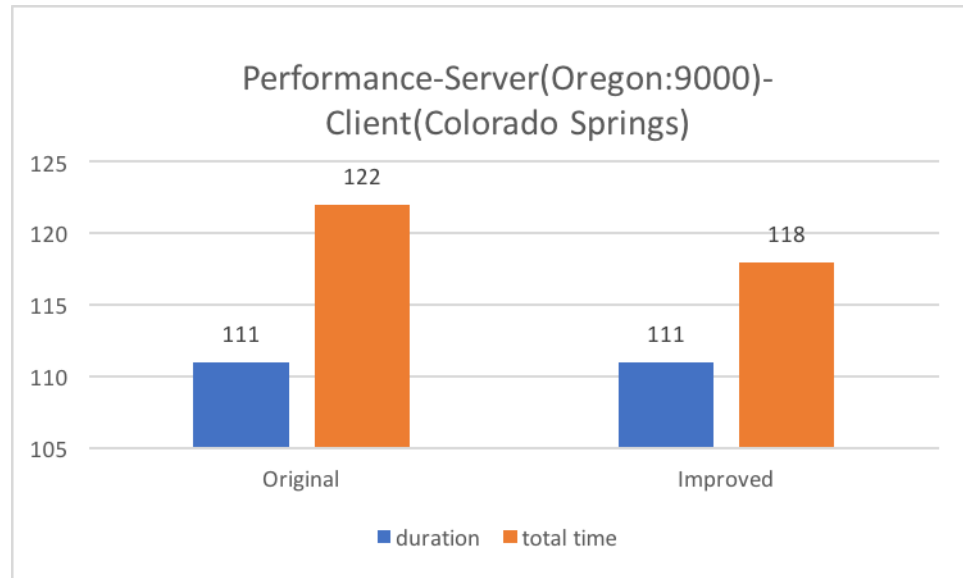
1. Request coming from a client located in Colorado Springs:



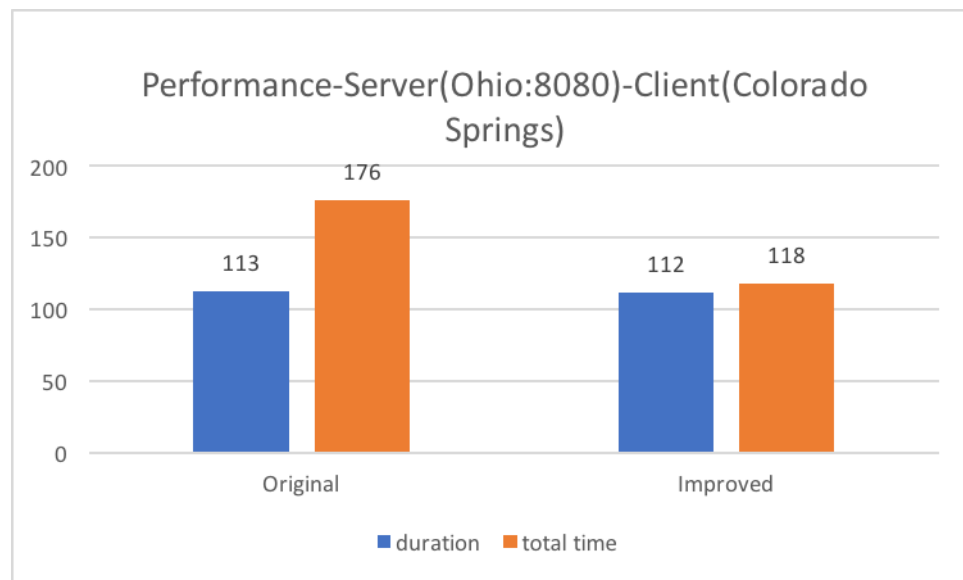
**Figure 3:** Performance from both the original and improved design for a client in Colorado Springs to a Server in EAS private cloud running on port 8000



**Figure 4:** Performance from both the original and improved design for a client in Colorado Springs to a Server in EAS private cloud running on port 9090.

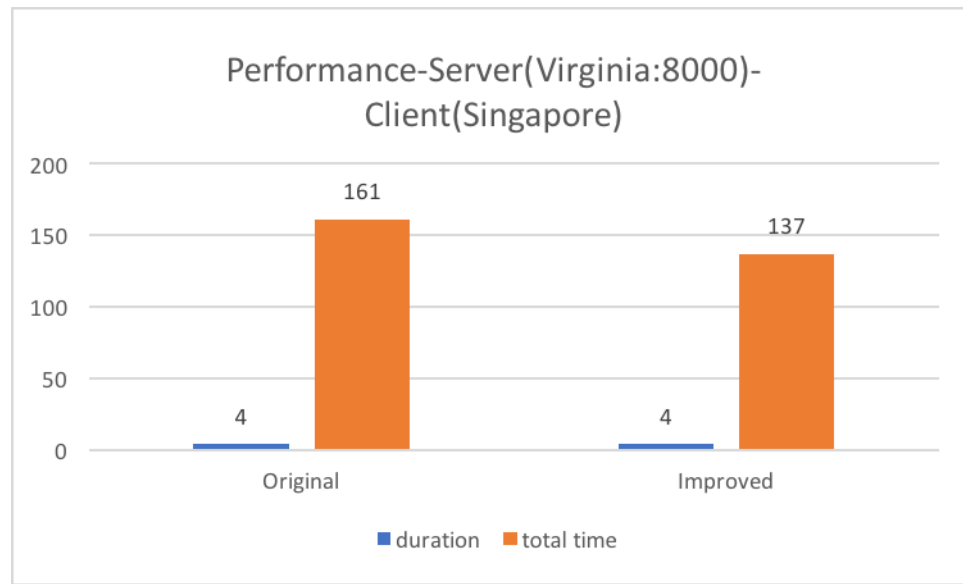


**Figure 5:** Performance from both the original and improved design for a client in Colorado Springs to a Server(Amazon Instance) in Oregon (West) region running on port 9000

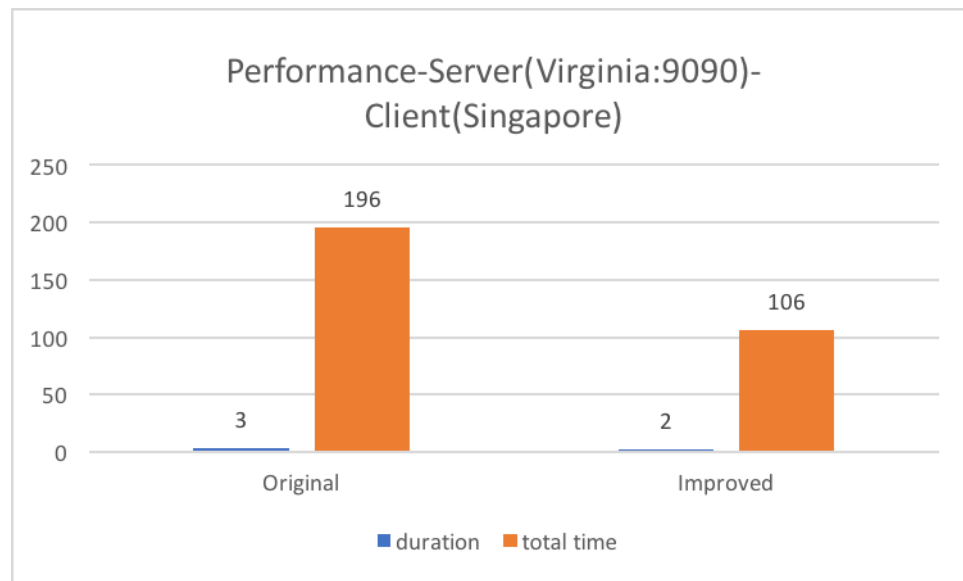


**Figure 6:** Performance from both the original and improved design for a client in Colorado Springs to a Server (Amazon Instance) in Ohio (East) region running on port 8080.

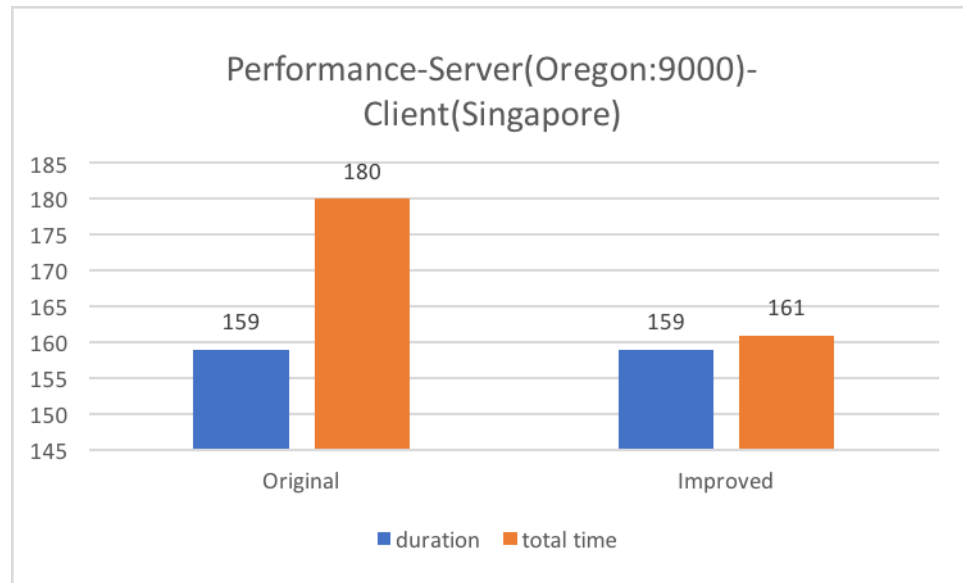
2. Request coming from a client located in Singapore:



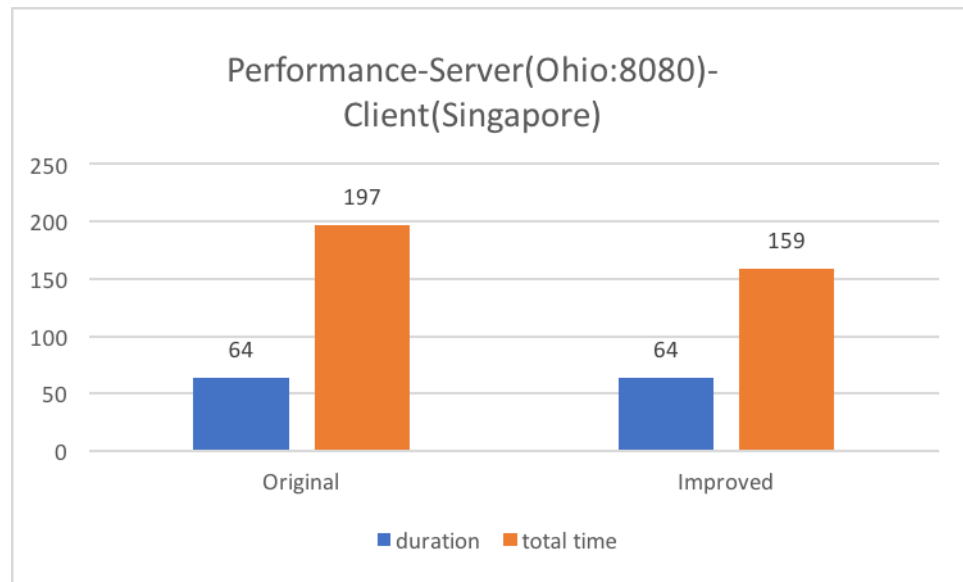
**Figure 7:** Performance from both the original and improved design for a client in Singapore to a Server(Amazon Instance) in N.Virginia (East) region running on port 8000



**Figure 8:** Performance from both the original and improved design for a client in Singapore to a Server(Amazon Instance) in N.Virginia (East) region running on port 9090.



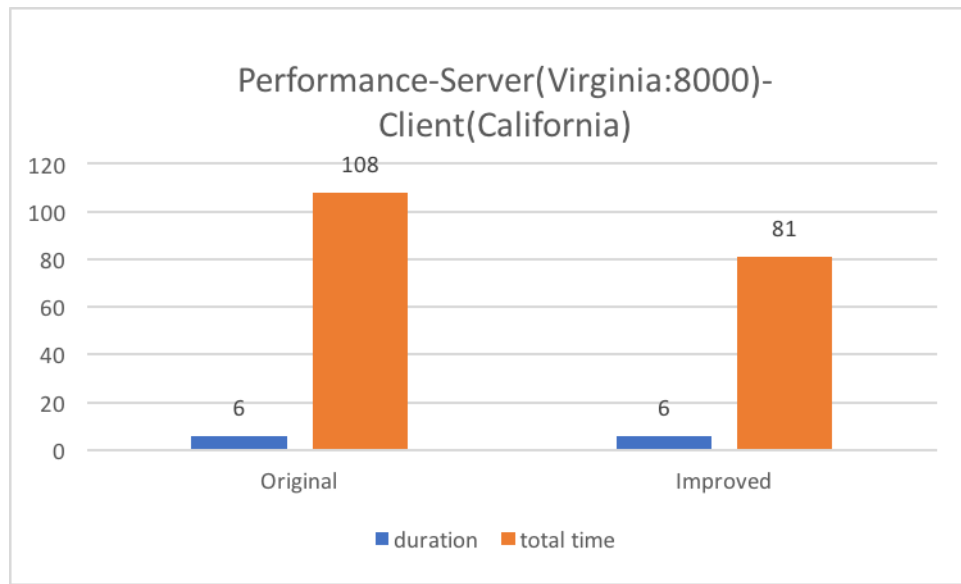
**Figure 9:** Performance from both the original and improved design for a client in Singapore to a Server (Amazon Instance) in Oregon (West) region running on port 9000.



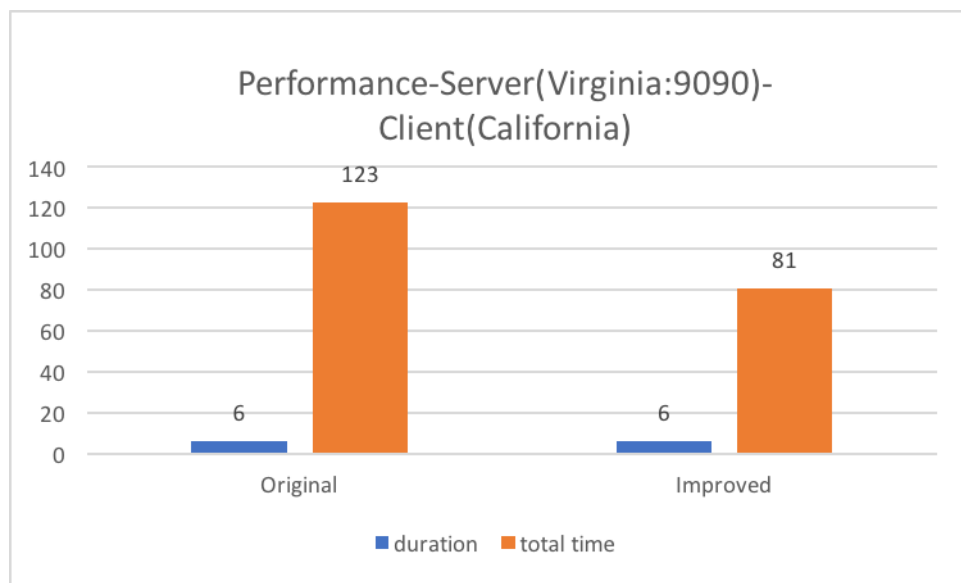
**Figure 10:** Performance from both the original and improved design for a client in Singapore to a Server (Amazon Instance) in Ohio (East) region running on port 8080.



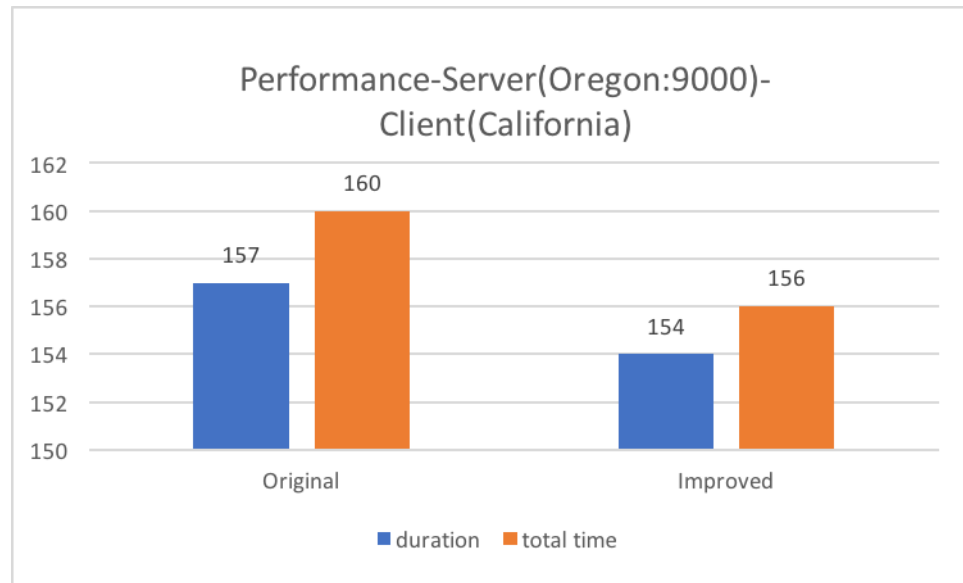
3. Request coming from a client located in North California:



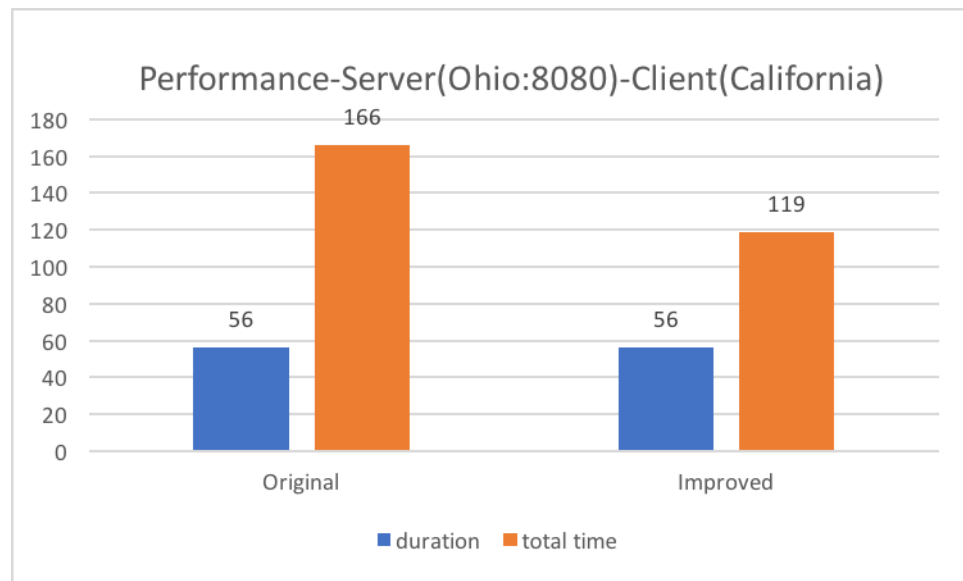
**Figure 11:** Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in N.Virginia (East) region running on port 8000.



**Figure 12:** Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in N.Virginia (East) region running on port 9090.

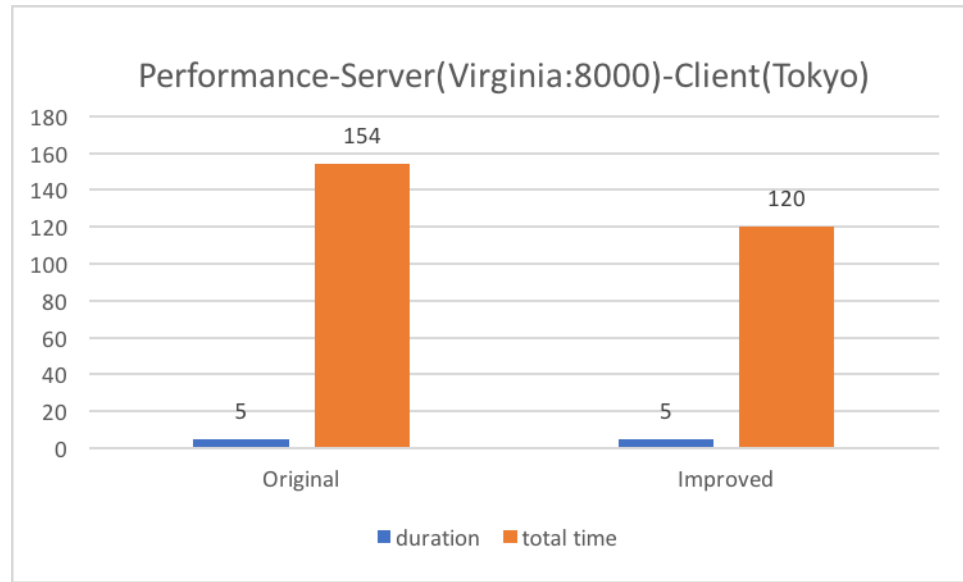


**Figure 13:** Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in Oregon (West) region running on port 9000.

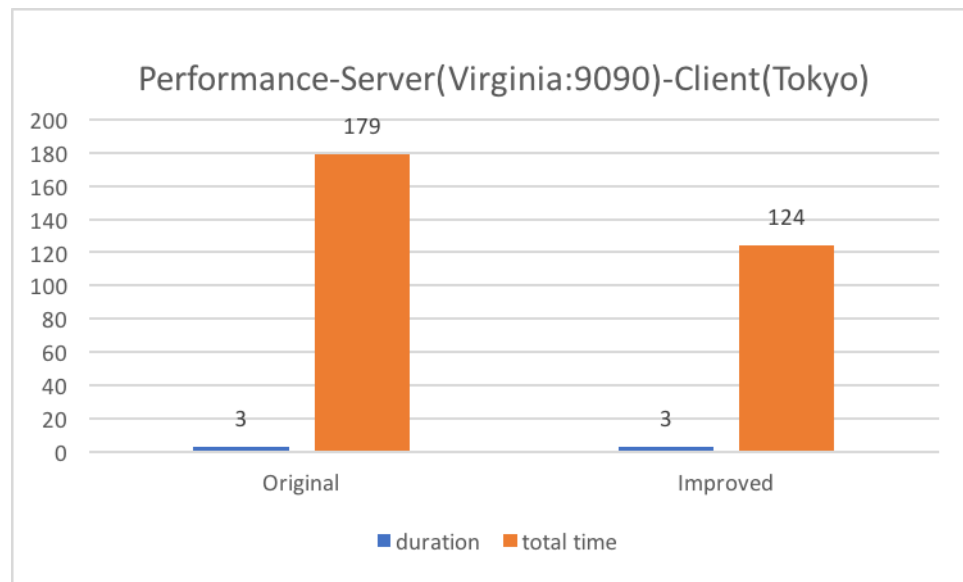


**Figure 14:** Performance from both the original and improved design for a client in California to a Server (Amazon Instance) in Ohio (East) region running on port 8080.

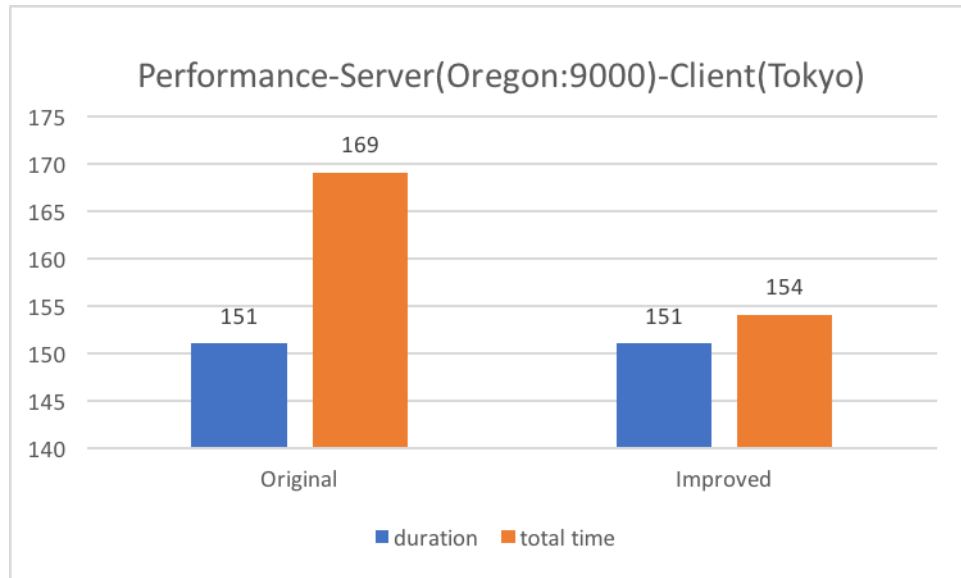
4. Request coming from a client located in Tokyo:



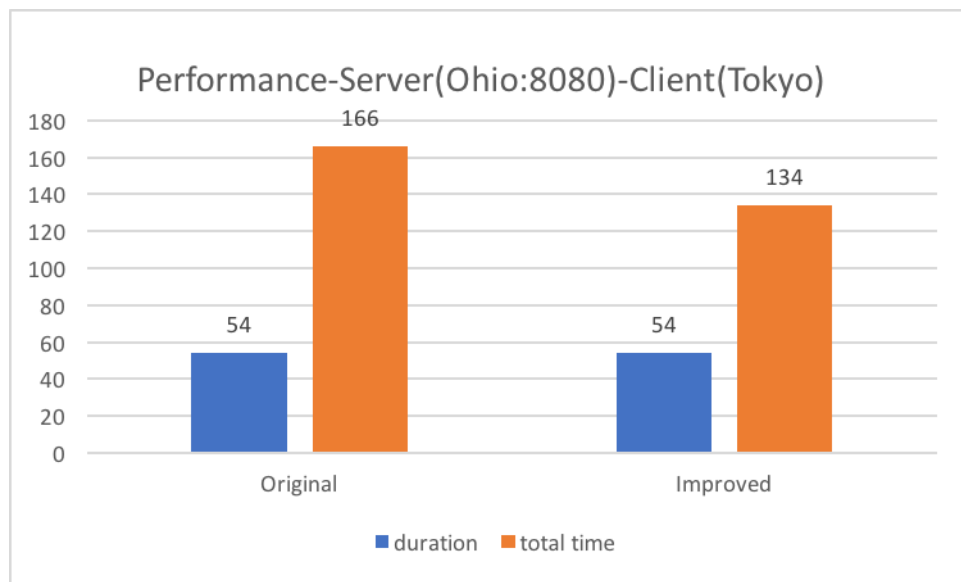
**Figure 15:** Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in N.Virginia (East) region running on port 8000



**Figure 16:** Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in N.Virginia (East) region running on port 9090



**Figure 17:** Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in Oregon (West) region running on port 9000.



**Figure 18:** Performance from both the original and improved design for a client in Tokyo to a Server (Amazon Instance) in Ohio (East) region running on port 8080.

We derive from our experiments and data, that more significant difference in the *total time* for delivery of the response to the client would be seen only in the web servers which are geographically placed much nearer to the client. The more the difference between the location, the less the significant difference in the *total time* for delivery of response to the client.

Even though, there is less difference between the *total time* of delivering the response to the client and the actual *duration* it took to retrieve the response within the server, it should be noted that 99 percent of the times, the *total time* from the improved design is still smaller than that from the original design. By this improvement in design we further achieved the desired resiliency for the web application performance.

### 5.2.2 Hops

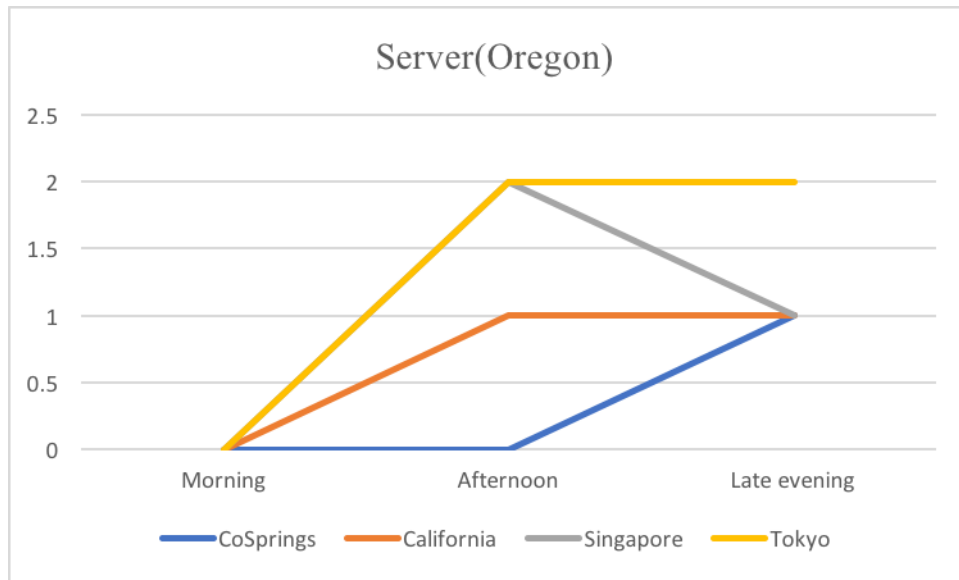
We calculated the ttl (Time-to-live) for each client to its servers' locations. The *time-to-live* (TTL) is the number of *hops* that a *packet* is permitted to travel before being discarded by a *router*. Ttl for clients from Singapore, N.California and Tokyo to servers in N.Virginia, Oregon and Ohio, and ttl for client from Colorado Springs to servers in EAS VI private cloud, Oregon and Ohio are as follows:

Client	N.Virginia	Oregon	Ohio	EAS VI
California	235	237	239	
Singapore	237	234	235	
Tokyo	234	233	234	
Colorado Springs		236	237	125

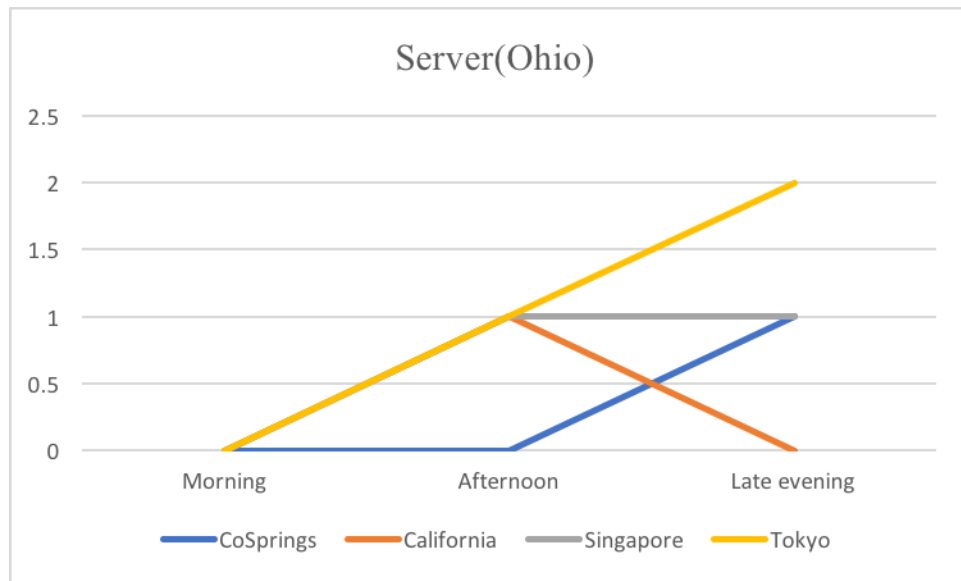
**Table 1:** Time-to-Live (ttl) readings from clients to all the available servers.

### 5.2.3 Time of the day

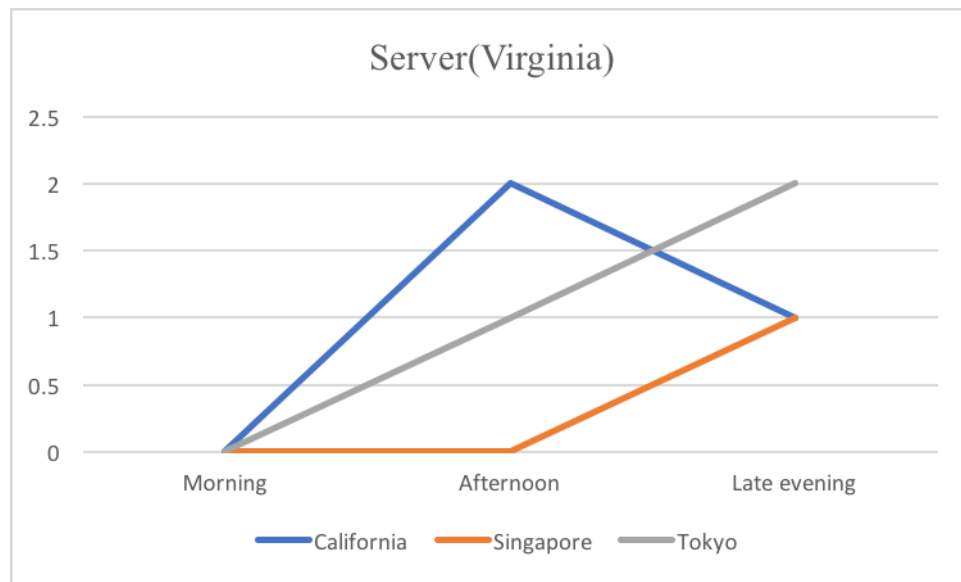
We recorded the responses' *total time* for all the clients to all the available servers in the mornings, afternoons and late evenings of the day. Typically, we recorded around 7:00am during the morning, 12:00pm during the afternoon and 10:00pm during the late evening. Our recorded data show that the *total time* is equal or comparatively less during the mornings than that of the late evenings and afternoons.



**Figure 19:** Differences in total time for clients to Oregon server, during different times of the day.



**Figure 20:** Differences in total time for clients to Ohio server, during different times of the day.



**Figure 21:** Differences in total time for clients to Virginia server, during different times of the day.

# CHAPTER VI

## LESSONS LEARNED & CHALLENGES FACED

### 6.1 Better Understanding of Node and packages

We learned a great deal about Node.js Web Servers. Being working with them, setting them up, linking from one server to another, helped me understand the functionality of this servers. A great effort of my study went into understanding how the packages work within them, once installed using npm (node package manager). I learned about some important packages in node which would make developing applications much easier. These include:

- **Express**

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. This package has a variety of HTTP utility methods and middleware to create an API easily. The HTTP utility methods include: GET, PUT, POST. The middleware include *route*, *set* and *use*. This package can be used to route HTTP requests (GET, POST, PUT), configure middleware, render HTML views and so on. We made use of this package for routing HTTP GET request from the client to the specified path with specified callbacks.

- **Simple-ssh**



This node package makes it easier to run a sequence of commands over SSH. This package is easier to install and use. We made use of this package to secure SSH over to the Amazon Instances. (Simple-ssh)

#### - **Child\_Process**

This is another node package we made use of. This package has several methods, including *exec* and *spawn*. We used *exec()* initially, but returned a buffered data; *exec()* should be used with caution as shell injection can be exploited. The *spawn* method spawns an external application in a new process and returns a streaming interface for I/O. We used this package's *spawn* method to start executing a web server as a child to our load balancer. The load balancer was able to start up a web server and was also able to kill that process when needed using *child.kill('SIGINT')* (*Child\_process*), where *child* is a *child\_process* instance.

#### - **Geoip-lite**

This is another node package we made use of for retrieving the longitude and latitudes of a given IP address. This API would return for a given IP address input, an object consisting of several properties, of which one of them is an array containing the longitude and latitude for the given IP address. (Geoip-lite)

## 6.2 Better Understanding of Promise

We made use of Promise instead of callback functions for asynchronous programming in our experiment. With Promises, we can resolve our data when and where it is needed, whereas with callbacks, the data has to be resolved then and there itself. This extra ability with Promises gave us a possible flexibility to use the data later in the time. We returned the responses from all the servers as Promises and once all promises are received, we then tried to resolve all of them together. Without Promises, we would not be able to achieve our desired functionality.

We even faced many challenges making promise work for us as to our desire. We resolved a promise prematurely and tried to send the response to the client after first server's response, which did not give us the opportunity to send other much faster response from another server to the client. After realizing the mistake, we resolved the promise much later than we did before, which gave us the desired result of having to sort them first based on time and then send the response to the client.

### **6.3 Better understanding of Amazon EC2 Instances**

With the help from Dr. Chow, I was able to set up Amazon Instances. I learned a great deal of installing Node in Instances, copying files/folders from desktop to the Instances, creating and assigning Elastic IP for each Instance and many more. Every action involving Instances is a good learning experience for us. We faced some challenges when dealing with Node.js servers in Amazon Instances. We initially used spawn to start the node server, but since we couldn't kill the process using `child_process_instance.kill('SIGINT')`, we had to lean on the simple-ssh API for secure connect to Amazon Instances and then execute the commands serially using its `exec()` functionality. We were able to start the servers within Instances using simple-ssh, but were not able to kill the processes when needed. We had to manually find out the process id for the running server and kill that process by streaming those command line arguments over the same shell, using `exec('kill $(pgrep -f app.js)')`.

## CHAPTER VII

### FUTURE DIRECTIONS

Our design can be expanded to accommodate the recording of durations from the servers, on high-level load balancer's side, from the previous sessions. Using these results, the high-level load balancer might effectively direct the clients to the optimal servers, for to serve the client in a much effective way, assuming *that* sever always serves the fastest response to the client. Also high-level load balancer can give the client the choice of redirect to a server based on its distance (using *geop-lite*) and/or servers' reported performance

Our client-side design can be improved much elaborately, in a way where dynamic interaction with the server-side would make the client side's application retrieve data dynamically. As for our existing design, we tested by only retrieving a file upon each *refresh* of the page i.e., upon each *get* request. We wish to extend our client-side to have applications where dynamic refresh of the content would be much suited. Such applications would benefit when used in a larger scale especially in coordination with web resiliency factor.

# CHAPTER VIII

## CONCLUSION

In this research, we developed a secure resilient web system with redundant Node.js web servers and multiple level of Node.js web clusters. The Node.js web servers are continuously monitored by the load balancer for their health and security. These multiple level of Node.js based load balancers are used to distribute requests to the backend web servers, start a web server, join this new server to the cluster. Depending on the origin of the requests, the load balancers will allocate or redirect them to the load balance clusters that are closer to the requested client.

By developing this secure resilient web system especially using Node.js web servers, we are able to serve a client faster with the response. The service to the client can never be disrupted even when a serving server becomes unresponsive. The high-level load balancer which takes upon it the whole responsibility of sending the request to the optimal server, based on location and throughput, would always serve the client with the response. This cloud based approach gives us the flexibility of reaching out for much reliable service. Therefore, based on the results of our experiment, we could conclude

that, a resilient web service can be offered to the clients using cloud based Node.js web servers.

# REFERENCES

[Ahma2015] R. W. Ahmad, A. Gani, S. H. A. Hamid, M. Shiraz, F. Xia, and S. A. Madani. Virtual machine migration in cloud data centers: a review, taxonomy, and open research issues. *The Journal of Supercomputing*, 71(7):2473–2515, 2015.

[Aias2014] M. Aiash, G. Mapp, and O. Gemikonakli. Secure live virtual machines migration: issues and solutions. In *Advanced Information Networking and Applications Workshops (WAINA)*, 2014 28th International Conference on, pages 160–165. IEEE, 2014.

[Alam2016] S. M. ALAmri and L. Guan. Exploring the firewall security consistency in cloud computing during live migration. In *Proceedings of the 7th International Conference on Computing Communication and Networking Technologies*, page 40. ACM, 2016.

[Alsh2016] H. Alshahrani, A. Alshehri, R. Alharthi, A. Alzahrani, D. Debnath, and H. Fu. Live migration of virtual machine in cloud: Survey of issues and solutions. In *Proceedings of the International Conference on Security and Management (SAM)*, page 280. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016.

[Anwar2013] M. Anwar. Virtual firewalling for migrating virtual machines in cloud computing. In *Information & Communication Technologies (ICICT)*, 2013 5th International Conference on, pages 1–11. IEEE, 2013. 8

[Nava2014] B. Navamani, C. Yue, X. Zhou, and E. Chow. An analysis of the virtual machine migration incurred security problems in the cloud. 2014.

[Voorsluys2009] Voorsluys, William, et al. "Cost of virtual machine live migration in clouds: A performance evaluation." *IEEE International Conference on Cloud Computing*. Springer Berlin Heidelberg, 2009.

[Tilkov2010] Tilkov, Stefan, and Steve Vinoski. "Node.js: Using JavaScript to build high-performance network programs." *IEEE Internet Computing* 14.6 (2010): 80-83.

[Teix2012] Teixeira, Pedro. *Professional Node.js: Building Javascript based scalable software*. John Wiley & Sons, 2012.

[Clark2005] Clark, Christopher, et al. "Live migration of virtual machines." *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005.

[Request] <https://www.npmjs.com/package/request>

[Amazon] <http://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/setting-up-node-on-ec2-instance.html>

[Bluebird] <http://bluebirdjs.com/docs/api-reference.html>

[ChildProcess] [https://nodejs.org/api/child\\_process.html#child\\_process\\_child\\_process\\_spawn\\_command\\_args\\_options](https://nodejs.org/api/child_process.html#child_process_child_process_spawn_command_args_options)

[Simple-ssh] <https://www.npmjs.com/package/simple-ssh>

[Geoip-lite] <https://www.npmjs.com/package/geoip-lite>

# APPENDIX A

**Installing node on Amazon Instances:** <http://docs.aws.amazon.com/sdk-for-javascript/v2/developer-guide/setting-up-node-on-ec2-instance.html>

## Testing Environment:

### 1. Web cluster in EAS VI private cloud (<http://viva.uccs.edu>):

#### 1. High-level load balancer:

- Node.js web server running on port 8080.
- Can be reached using: <http://viva.uccs.edu:8080>.
- Set up environment can be found at:  
<http://viva.uccs.edu/~pgunnam/thesis/vm1/nodejs/>
- Server set up-Original design (Simple-ssh): app-simple-ssh.js
- Improved design : app-simple-ssh-alt-1.js
- To start, execute: node app-simple-ssh.js or node app-simple-ssh-alt-1.js

#### 2. WebServer-1:

- Node.js web server running on port 8000.
- Can be reached: <http://viva.uccs.edu:8000>.
- Set up environment can be found at:  
<http://viva.uccs.edu/~pgunnam/thesis/vm2/nodejs/>
- Server set up: http.js
- To start, execute: node http.js



### 3. WebServer-2:

- Node.js web server running on port 9000.
- Can be reached: <http://viva.uccs.edu:9000>.
- Set up environment can be found at:  
<http://viva.uccs.edu/~pgunnam/thesis/vm3/nodejs/>
- Server set up: http.js
- To start, execute: node http.js

### 4. WebServer-3:

- Node.js web server running on port 9090.
- Can be reached: <http://viva.uccs.edu:9090>.
- Set up environment can be found at:  
<http://viva.uccs.edu/~pgunnam/thesis/vm4/nodejs/>
- Server set up: http.js
- To start, execute: node http.js

## 2. Web cluster in Amazon Ohio (East) region

### 1. Low-level load balancer:

- Node.js web server in Amazon Linux environment running with Elastic IP: 52.14.178.188.
- Port: 8080
- Can be reached: <http://52.14.178.188:8080>
- Set up environment can be found at:  
<http://52.14.178.188:8080/server1/nodejs/>
- Server set up: app.js
- To start, execute: node app.js

## 2. WebServer-1:

- Node.js web server in Amazon Linux environment running with Elastic IP: 52.14.188.220.
- Port: 8000
- Can be reached: <http://52.14.188.220:8000>
- Set up environment can be found at:  
<http://52.14.188.220:8000/server2/nodejs/>
- Server set up: http.js
- To start, execute: node http.js

## 3. Web Server in Amazon Oregon (West) region:

- Node.js web server in Amazon Linux environment running with Elastic IP: 54.69.97.75.
- Port: 9000
- Can be reached: <http://54.69.97.75:9000>
- Set up environment can be found at:  
<http://54.69.97.75:9000/server/nodejs/>
- Server set up: http.js
- To start server, execute: node http.js

## 4. Web cluster in North Virginia substituting for EAS VI high-level load balancer:

### 1. High-level load balancer:

- Node.js web server running on port 8080.
- Can be reached using: http://34.200.60.134:8080.
- Set up environment can be found at: <http://34.200.60.134/vm/nodejs/>
- Server set up-Original design: app-simple-ssh.js
- Improved design: app-simple-ssh-alt-1.js

- To start, execute: `node app-simple-ssh.js` or `node app-simple-ssh-alt-1.js`

## 2. WebServer-1:

- Node.js web server running on port 8000.
- Can be reached: <http://34.200.60.134:8000>.
- Set up environment can be found at: <http://34.200.60.134/vm1/nodejs/>
- Server set up: `http.js`
- To start, execute: `node http.js`

## 3. WebServer-2:

- Node.js web server running on port 9090.
- Can be reached: <http://34.200.60.134:9090>.
- Set up environment can be found at: <http://34.200.60.134/vm2/nodejs/>
- Server set up: `http.js`
- To start, execute: `node http.js`

## Third Party Libraries:

- `simple-ssh` <https://www.npmjs.com/package/simple-ssh> for SSH to Amazon Instances.
- `bluebird` <http://bluebirdjs.com/docs/getting-started.html> for Promise.
- `request` <https://www.npmjs.com/package/request> for *get* request.
- `geoip-lite` <https://www.npmjs.com/package/geoip-lite> for retrieving longitude and latitude of a given IP address.

## Development Tools:

- Firefox or Safari for client-side browser (for Colorado Springs client) and amazon console for other clients from California, Singapore and Tokyo.
- Linux console for debugging.

# APPENDIX B

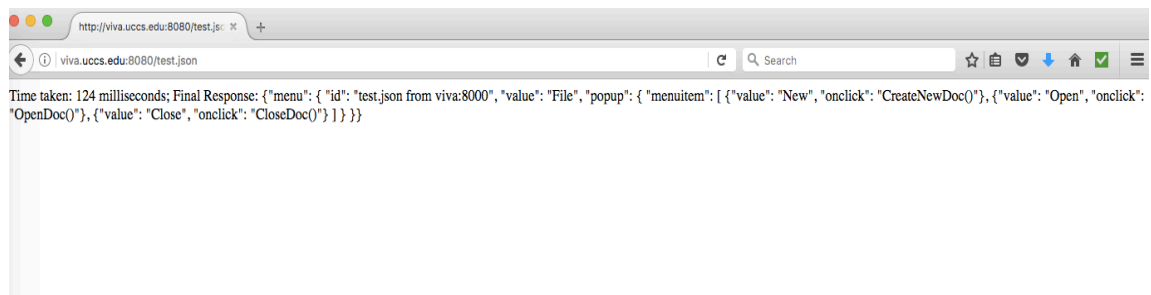
## Demo

### a. Request from a client from Colorado springs:

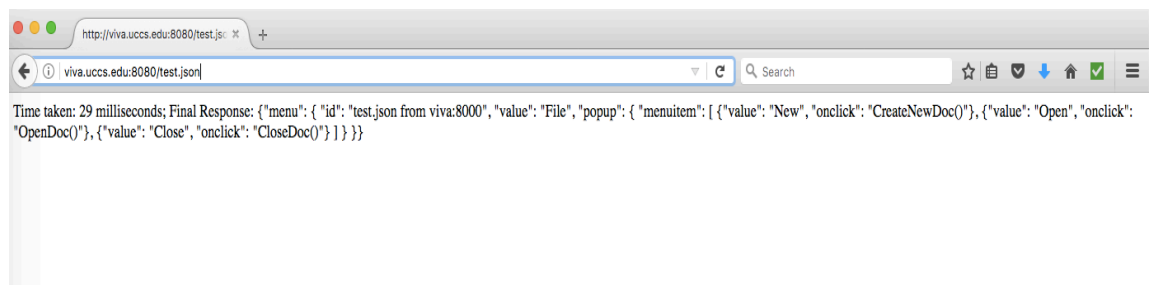
1. When all the servers are responsive, the client receives the response from

<http://viva.uccs.edu:8000/>

#### Original Design:

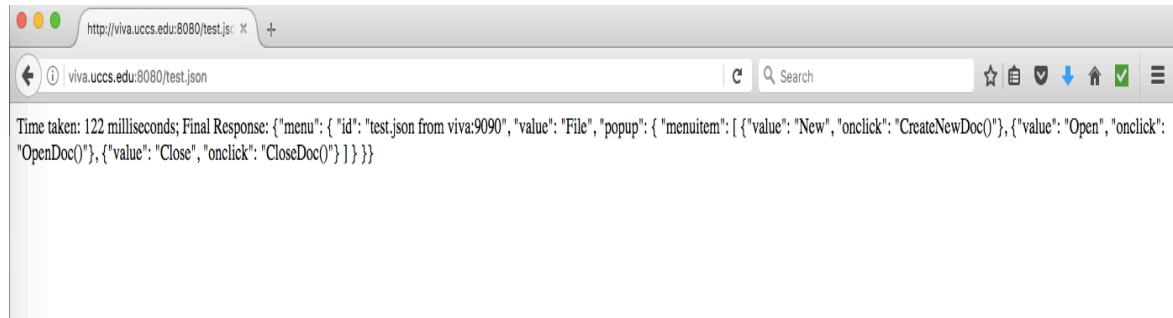


#### Improved Design:

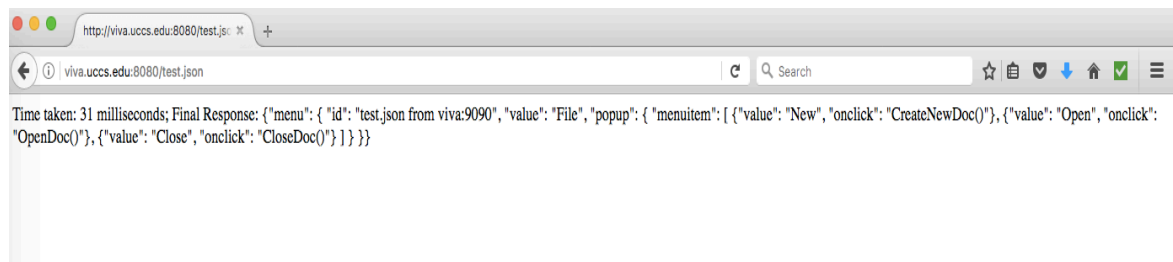


2. When all the servers except `viva.uccs.edu:8000` are responsive, the client receives the response from <http://viva.uccs.edu:9090/>

### Original Design:

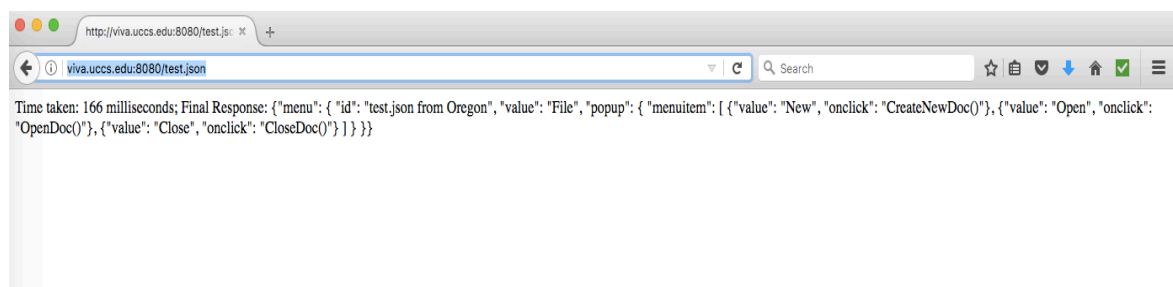


### Improved Design:

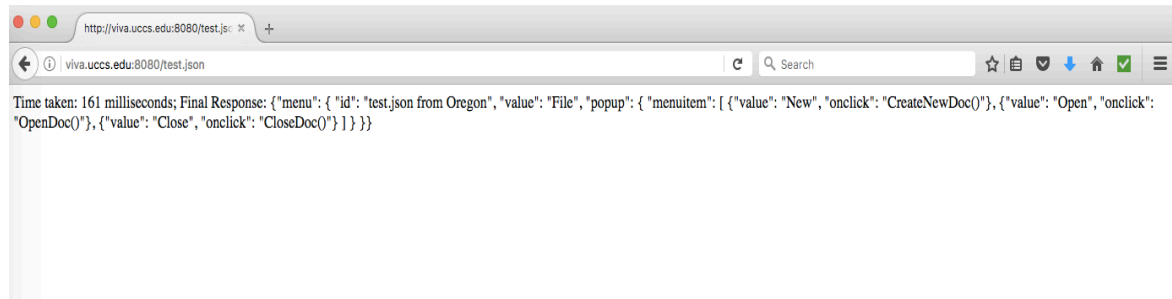


3. When all the servers other than those within EAS VI private cloud are responsive, the client receives the response from Oregon Amazon Instance:

### Original Design:



### Improved design:



### b. Request from a client from Tokyo:

When all the servers are responsive, the client from Tokyo receives the response from

Virginia Instance (high-level load balancer):

### Original Design:

```
pragati — ec2-user@ip-172-30-0-65:~ — ssh pgunnam@viva.uccs.edu — 80x13

[ec2-user@ip-172-30-0-65 ~]$ wget http://34.200.60.134:8080/test.json
--2017-05-07 21:59:46-- http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 309 [text/html]
Saving to: 'test.json.26'

test.json.26      100%[=====>]      309  --.-KB/s    in 0s

2017-05-07 21:59:46 (33.4 MB/s) - 'test.json.26' saved [309/309]

[ec2-user@ip-172-30-0-65 ~]$

pragati — ec2-user@ip-172-30-0-65:~ — ssh pgunnam@viva.uccs.edu — 114
Time taken: 188 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:9090",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

### Improved Design:

```

pragati — ec2-user@ip-172-30-0-65:~ — ssh pgunnam@viva.uccs.edu — 80x13

[ec2-user@ip-172-30-0-65 ~]$ wget http://34.200.60.134:8080/test.json
--2017-05-07 22:08:24-- http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 307 [text/html]
Saving to: 'test.json.28'

test.json.28      100%[=====>]      307  --.-KB/s    in 0s

2017-05-07 22:08:25 (70.1 MB/s) - 'test.json.28' saved [307/307]

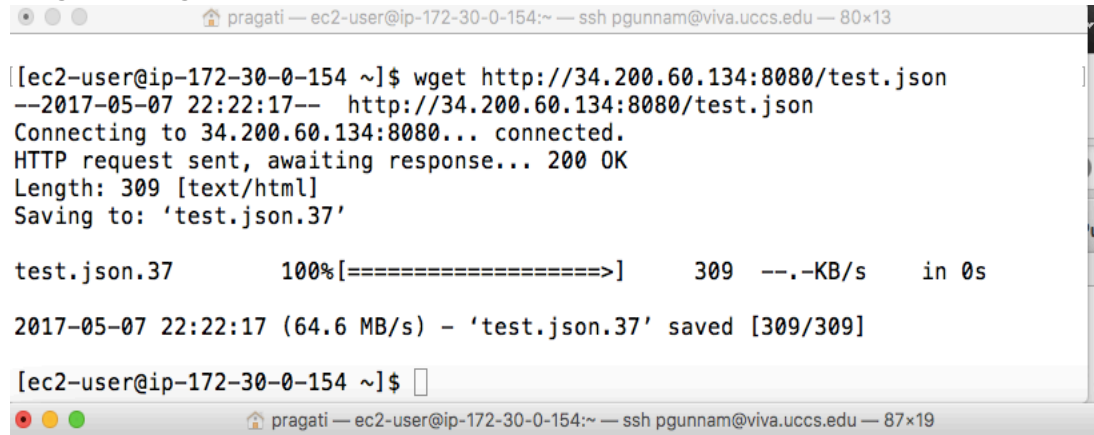
[ec2-user@ip-172-30-0-65 ~]$ 
pragati — ec2-user@ip-172-30-0-65:~ — ssh pgunnam@viva.uccs.edu — 114x
Time taken: 32 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:8080",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}
~
~
~
~
~

```

c. **Request from a client in Singapore:**

When all the servers are responsive, the client in Singapore receives the response from Virginia Instance (high-level load balancer):

**Original Design:**



```
[ec2-user@ip-172-30-0-154 ~]$ wget http://34.200.60.134:8080/test.json
--2017-05-07 22:22:17--  http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 309 [text/html]
Saving to: 'test.json.37'

test.json.37      100%[=====>]      309  --.-KB/s   in 0s

2017-05-07 22:22:17 (64.6 MB/s) - 'test.json.37' saved [309/309]

[ec2-user@ip-172-30-0-154 ~]$
```

```
Time taken: 169 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:9090",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
```

```
~
~
~
~
~
~
~
"test.json.37" 11L, 313C
```



## Improved Design:

```

pragati — ec2-user@ip-172-30-0-154:~ — ssh pgunnam@viva.uccs.edu — 80x13

[ec2-user@ip-172-30-0-154 ~]$ wget http://34.200.60.134:8080/test.json
--2017-05-07 22:25:50-- http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 307 [text/html]
Saving to: 'test.json.39'

test.json.39      100%[=====>]      307  --.-KB/s    in 0s

2017-05-07 22:25:51 (69.5 MB/s) - 'test.json.39' saved [307/307]

[ec2-user@ip-172-30-0-154 ~]$ 
pragati — ec2-user@ip-172-30-0-154:~ — ssh pgunnam@viva.uccs.edu — 87x19
Time taken: 29 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:9090",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
~
~
~
~
~
~
~
"test.json.39" 11L, 311C
1,1

```

**d. Request from a client in N.California:**

When all the servers are responsive, the client in California receives the response from Virginia Instance (high-level load balancer):

**Original Design:**

```

pragati — ec2-user@ip-172-30-0-146:~ — ssh pgunnam@viva.uccs.edu — 80x13
sys      0m0.000s
[ec2-user@ip-172-30-0-146 ~]$ time wget http://34.200.60.134:8080/test.json
--2017-05-07 22:31:00--  http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 309 [text/html]
Saving to: 'test.json.20'

test.json.20      100%[=====]      309  --.-KB/s    in 0s

2017-05-07 22:31:00 (62.3 MB/s) - 'test.json.20' saved [309/309]

pragati — ec2-user@ip-172-30-0-146:~ — ssh pgunnam@viva.uccs.edu — 87x19
Time taken: 195 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:8080",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
~
~
~
~
~
~
~
"test.json.20" 11L, 313C
2,3

```

### Improved Design:

```

pragati — ec2-user@ip-172-30-0-146:~ — ssh pgunnam@viva.uccs.edu — 80x13

[ec2-user@ip-172-30-0-146 ~]$ wget http://34.200.60.134:8080/test.json
--2017-05-07 22:37:02-- http://34.200.60.134:8080/test.json
Connecting to 34.200.60.134:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 307 [text/html]
Saving to: 'test.json.25'

test.json.25      100%[=====>]      307  --.-KB/s    in 0s

2017-05-07 22:37:02 (74.7 MB/s) - 'test.json.25' saved [307/307]

[ec2-user@ip-172-30-0-146 ~]$
pragati — ec2-user@ip-172-30-0-146:~ — ssh pgunnam@viva.uccs.edu — 87x19
Time taken: 34 milliseconds; Final Response: {"menu": {
  "id": "test.json from virginia:8000",
  "value": "File",
  "popup": {
    "menuitem": [
      {"value": "New", "onclick": "CreateNewDoc()"},
      {"value": "Open", "onclick": "OpenDoc()"},
      {"value": "Close", "onclick": "CloseDoc()"}
    ]
  }
}}
~
~
~
~
~
~
~
"test.json.24" 11L, 311C
2,3

```