Evaluate scalable and high-performance Node.js Application Designs

Advanced Web Technology Project

Submitted to: Martin Lasak

Hafiz Umar Nawaz
Computer Science
TU Berlin
h.umarnawaz@gmail.com

Denisa Rucaj
Computer Science
TU Berlin
rucaj.d@gmail.com

Naveed Kamran
Computer Science
TU Berlin
naveedkamran4@gmail.com

***Abstract*—The goal of this project is to measure the impact of different application architectures and designs on scalability of a simple Node application. We researched about different scalable application architectures. We analyzed the impact of different components in system in terms of performance and scalability. In this project we used simple REST services built with Node.js. We deployed our service on different instance configurations. For example, a single machine instance or multiple host machines using docker and nginx load balancer. Generally, the performance metrics we considered for our experiments are latency and throughput. We also carefully observed about elastic scalability, and error rate.**

Keywords—scalability. software architecture, software design, get, insert, throughput, latency, error rate, load balancing.

I. Introduction

Nodejs has gained a lot of popularity in last few years, because of being lightweight and rapid application development. Another major reason for its popularity is its based on ECMAScript 2015 (ES6) standard, which means that javascript developers can write server side code as well.

Scalability is defined as number of transactions or user requests processed concurrently without any delay or degradation in service [1].

According to Stefan Tai, David Bermbach and Erik Wittern [1], scalability is a general property of a system or service, which describes whether said system compared to a base configuration able to do handle more requests when deployed on more resources. That is, scalability describes the relationship between the change in available resources and the resulting change in provided computation.

Elasticity, on the other hand, describes what happens during the period of adding or removing computing resources (for example, processing power, bandwidth, hard disk space or memory capacity) how long does it take the system to adapt to the new state and what is the impact on other qualities, e.g., performance, in the meantime.

Scalability can be achieved in two ways:

Scaling up / vertical scalability - Scaling up means upgrading the current capabilities of computing resource. For example, adding more hard disk space, memory, bandwidth or computing power to existing node. Usually, scaling up is expensive after a certain point of line. For example, computation power could be increased up to a certain level and after that level either it becomes very expansive or limit of device capabilities comes into play. For example, CPU computation power can not grow beyond a certain limit.

Scaling out or horizontal scalability - Scaling out means adding more resources to the cluster. For example, new nodes. Scaling out is comparatively easy to achieve as compared to vertical scalability. Horizontal scalability could be achieved by simply adding more nodes into the cluster. Horizontal scalability is also inexpensive as compared to vertical scalability.

Horizontal scalability can ensure higher scalability. However, multiple nodes introduce complexity and some problems as well. For example we have to take care of following questions:

How to distribute the requests to different nodes?

1. How to ensure state synchronization? State synchronization involves data synchronization and user session status synchronization etc.
2. What if one node goes down? For example, we can have master slave nodes. Master node can listen the heartbeat of slave nods and if some slave node goes down then master can either prepare another node to handle the requests or can simply give the workload to another node. But, what if the master node goes down? In this case, some

slaves node needs to start behaving like a master. This is a bit complex approach, but ensure high scalability.

Simply adding new resources does not always ensure high scalability and elasticity. The new nodes need to be added intelligently, such that more scalability could be achieved.

Other than the dimensions of scalability, here are different algorithms or methods for achieving a scalable system:

1. **Co-location** - It means existence of data in a physically connected and accessible space. If the data is co-located then it is fast to access. For example, co-location of data in memory or hard disk matters a lot. If data is dispersed then latency can occur because of searching for data onto another place. For example, in case of reading data from hard disk, it involves the hard disk read head to move to proper sector and track.
2. **Caching** - Caching means, making the frequently used data available from a faster memory access. Caching helps to avoid latency and hence, helps in achieving scalability. For example, if some data structure usually needs to be read from hard disk then we can keep it in memory such that it does not need repeated reading from hard disk.
3. **Divide and conquer** - This approach is useful when we have multiple compute resources. We can divide the processing requests to be done on different compute resources.

In this project, we have explored different options to make Node.js applications more scalable. Therefore, we performed different scalability benchmarking and checked how it would affect the elasticity of the application. We tested scalability on a very simple RESTful API Node.js application. The scalability benchmarking tests include two different aspects of testing:

1. Testing an application that does not involve any database, but some computations were being performed in memory.
2. Testing an application that have a database. Database involvement means also assuring that the database system scalability matches the application scalability.

A rest application performance refers to the quality and efficiency (for a given set of hardware configurations, how many requests could be handled) at which an application functions. Many factors can affect application performance. For example, bandwidth capacity, the number of concurrent users on a network, application protocols and application architecture itself.

For benchmarking the performance of the application we have used following metrics:

1. **Throughput** (Measured in number of requests handled per second). While measuring throughput we took into account that if even a single request fails, then we will consider that scenario as failed.
2. **Latency** (Latency is measured as the average time that it takes to respond a request). Low latency shows better performance.
3. **Error rate** (If we have any single error then we considered that scenario as failed). Error could occur because of non-availability of application, or application can not handle the current workload or, because of a runtime error or response is not received within a specified time period (response timeout error).

We used JMeter to perform the tests. JMeter can tell the throughput for a test, but to measure the latency we need to find it from JMeter execution logs.

II. Related Work

Roy T. Fielding, and Richard N. Taylor have built an architecture for IP/TCP/HTTP Redirection Based Approaches, more straightforward, is shown how the typical web applications architectures look like.

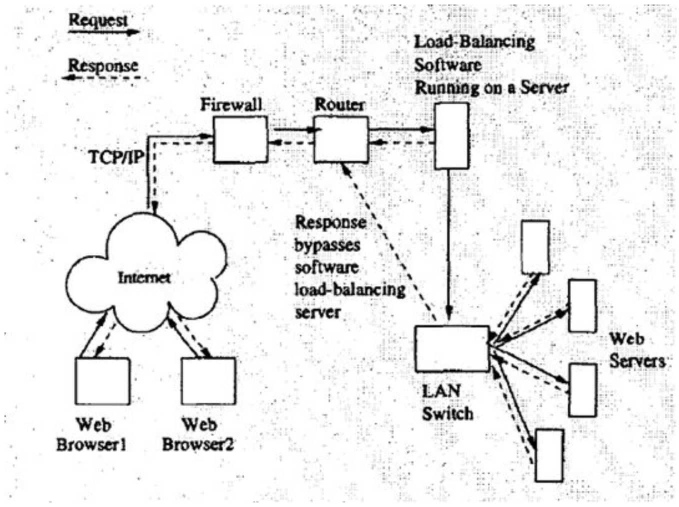

Request
Response
Load-Balancing Software Running on a Server
Firewall
Router
TCP/IP
Response bypasses software load-balancing server
Internet
Web Servers
LAN Switch
Web Browser1
Web Browser2

Fig. 1. Web Server Farm with Hardware Load-Balancing

Server [2].

In this architecture we have multiple web services which are connected through simple networking interface. The user request comes to load balancer after passing through firewall and network router. The load balancer decides which web server needs to respond for the requested resource.

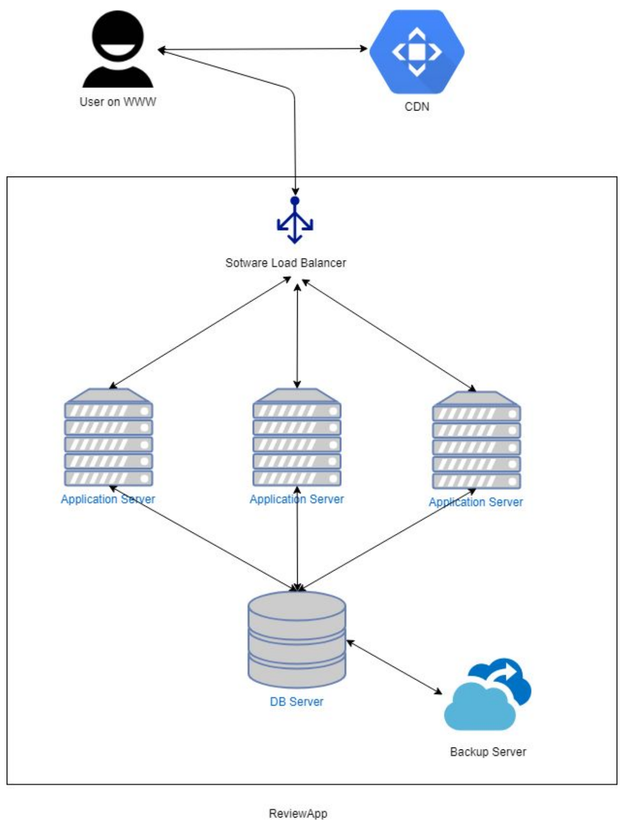


Fig. 2. Single database and multiple application servers [3].

This architecture is based on single database server and multiple application servers. The data come from external resources and static content. The application server could be added or removed depending upon the workload. The application servers retrieve and send data to the same Database storage, which has one backup server, in case of downtime, fire or nature accidents. The common resources could be deployed on a Content Delivery Network (CDN) to avoid load on major instances. Our application is a simple REST application and it does not have any resources to be deployed on a CDN. Hence, we have not used CDN, but the CDN shown here is just to elaborate the architecture.

All the external requests received are directed to the software based load balancer which is responsible for keeping heartbeat information of all running nodes. The load balancer selects one of the node that needs to handle the request. Node selection can be based on many parameters. For example:

1. Least connections to decide the minimum load per node [4].
2. Least time, favours the servers with the lowest average response times [4].
3. Applying round robin to decide in what way the load will be sent to the nodes [4].
4. Generic hash or IP hash, Using hashing function to decide a node. Hashing could be based on request IP address, for example [4].
5. Randomly selecting a node
6. Leader election algorithm, which is a process used to assign, via network a single process as the organizer or leader of some tasks distributed among several computers (nodes) [4].

III. Our Approach

To test the scalability of the application we have developed a small Review Application. We created a simple reviews application that was build using MongoDB, NodeJS, ExpressJS, Simple Frontend. It stores rating, comments text, timestamps.

To be able to scale on different hosts, one scenario would be to deploy databases on different machines from application instances. Initially, were considered two databases for the Review application, Apache Cassandra and MongoDB. Below, is given a short description related to how the scalability and high performance applies on them.

Cassandra is a column-based-store database. It stores the column name with each data records, and the name of columns can include data. A column in Cassandra consists of column_name, value and timestamp. It is based on Multi Data Replication and Virtual Nodes. Apache Cassandra has an architecture adapted for scalability and high performance and is best used for very large data [8].

MongoDB is a document-based-store database. It scales horizontally using sharding. Sharding is a horizontal partitioning of data. The user chooses a shard key, which determines how the data in a collection will be distributed. The data is split into ranges and distributed across multiple shards. MapReduce Aggregation function can be used for batch processing of data and aggregation operations. MongoDB provides high-availability with replica sets [9].

To narrow the scope of the project, and due to limited time, we decided to implement in our application only one database and specifically, MongoDB. MongoDB package for Node.js has a huge community support, that makes it easier to be implemented on the Node.js applications.

So this way, was used only one database server. Though the single database server is the single point of failure, but the purpose of this project was to learn about the scalability of the Node.js, and not the database server. That is why we did not put much time on database server.

These were all possibilities, as mentioned in section II, to use a load balancing method. But we have used least connections and round robin algorithm, because

- Its a default method in NGINX and does not need any extra configurations to make it working.
- It keeps all the nodes busy.

As our GET and POST requests almost take the same processing power which means that if we divide the requests to the existing computation resources then it must complete almost in same time. So, we don't really need to worry about finding the least busy nodes and sending request to that node.

IV. Setup

Hardware Configurations used for Experimentation:

CPU: Core i5 @ 2.6 GHz
RAM: 16 GB
Ubuntu 16.04, 16 Bit OS
Network: 100 Mbps

The application we created is simple Restful application that creates (Http request type POST) and reads (Http request type GET) review requests, which contain a text comment and a rating number from 1 - 5. Every request is recorded by a request id. After the request is posted or created, the information is sent back as JSON data.

We followed the guides from this two resources [6], [7], [10] , [11].

Code Organization

```
.
├── app.js
├── docker-compose.yml
├── Dockerfile
├── nginx
│   ├── Dockerfile
│   ├── localhost.error_log
│   └── nginx.conf
├── node_modules
├── package.json
├── package-lock.json
├── README.md
├── resc
│   ├── presentations
│   └── testing
├── review
│   └── ReviewController.js
└── src
    └── api
    ├── docker-compose.yml
    ├── nginx
    │   └── nginx.conf

├── node_modules
└── review
        └── ReviewController.js
```

About mongodb docker container

A volume parameter, which we have called data-volume, is instantiated to store our mongo files, to ensure that the data exceed even after the mongo container is deleted. This parameter maps to the created data-volume in /data/db folder where the mongo storage file rests.

About nginx.conf

Upstream is a module used by NGINX to load balance over HTTP servers when using NGINX HTTP module. The upstream module also defines how any individual request is assigned to any of the upstream servers. We have defined three upstream server. The reverse proxy is placed on front of the Node.js server, defined with proxy_pass: http://node-app. It is used to prevent Node.js server from direct internet traffic and allows flexibility using multiple servers in load balancing across the servers and in caching content.

Using NGINX as Node.js reverse proxy has following advantages for our application [4]:

1. Managing Node.js crashes elegantly
2. Simplifying privilege handling and port assignment.

least_conn is an NGINX load balancing algorithm. For each upstream application servers the max_fails directive is set to 3 and fail_timeout to 30 sec and the weight parameter instructs NGINX to pass at least 10 connections to each server.

proxy_cache_bypass There are many scenarios that demand that the request is not cached. For this, NGINX exposes a proxy_cache_bypass directive that when the value is non empty or non zero, the request will be sent to an upstream server rather than be pulled from cache [4].

Below, are the different architecture scenarios we used for experimentation.

First application architecture -- Single Node Application Architecture

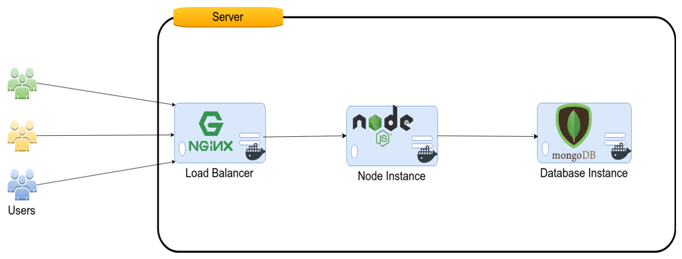


Fig. 3. Architecture with single Node.js instance

Second Application Architecture -- Multiple Node Instances on Single Machine

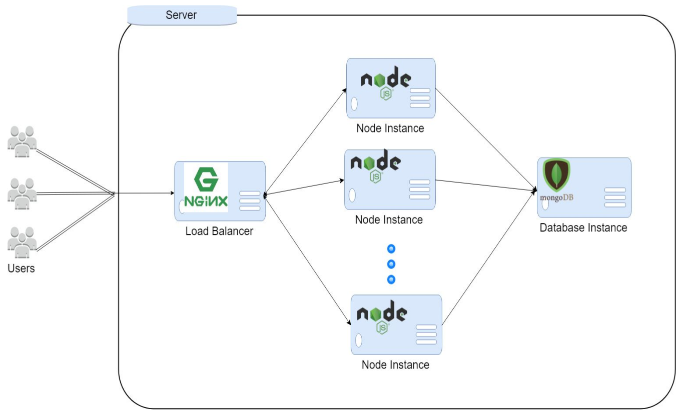


Fig. 4. Application with multiple Node.js instances

In the architectures shown in Fig. 3., Fig. 4., Fig. 5. we have:

1. Nginx Instance: an instance for Nginx used as a load balancer.
2. Node Application Instances: there were several instances of our reviews NodeJS application. One instance in Fig. 3., three to seven instances in Fig. 4., Fig. 5..
3. Database Instance: an instance for mongodb.

Each of the instances is running in separate Docker containers [11]. In front of the Node.js application instances, is build a single NGINX instance. The NGINX instance will reverse proxy to the application instance and will load balance through port 80 using a load balancer round robin fashion and least connection algorithm. NGINX image runs in a separate Docker container and it links to the other containers, Node.js containers and MongoDB container. Docker compose compose the application linking the containers. Each node is build on mongo environment sending and retrieving data to reviewdb through the port 27017. The images of containers are build from pre-built docker images in Docker Hub. To build the containers we have created one Dockerfile for each container.[4]

There can be multiple clusters to handle a request from user. Every instance is allowed to use all the available system resources at any time. Hence, we did not restricted our instances for any type of resources, for example, memory, disk space, bandwidth, CPU cycles.

Third application architecture -- Multiple Node Containers on Multiple Physical Machines

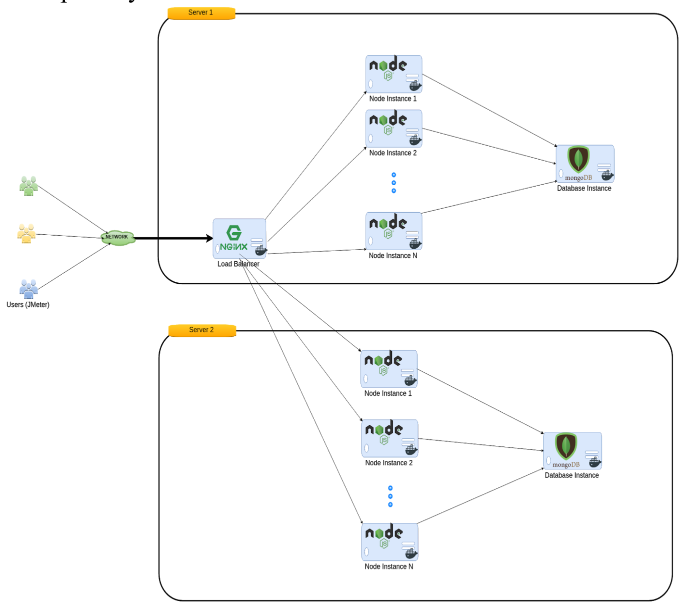


Fig. 5. Architecture with physical multiple servers

The request is handled as below:

1. A user from internet accessed the IP address or URL of our application.
2. The request comes to Load balancer which is responsible to route the request to proper NodeJS node.
3. If a request is forwarded to NodeJS node it is responsible to handle the request. Every node instance is connected to mongodb using TCP IP. If a request could not be handled then appropriate HTTP

status code is returned such that the client knows about the error type.
4. If all the nodes are busy and there is no capacity to handle a new request then the web server responds with timeout http response.

V. Repeating Experiment

To repeat the experiments, we need to build the re-environment. In abstract, following things need to be repeated:

1. Setup Docker and Docker Compose
2. Setup Application. Application setup instructions could be found in readme file on GitHub.
3. Setup Docker instance monitoring environment (CAdvisor). (This step is not mandatory and could be skipped).
4. Setup JUnit Testing Framework. JUnit project is also available in GitHub under resc folder.
5. Perform Testing

VI. Project Management

For managing our project we used a combination of project management tools.
We used Asana to create, sign to different members , set a deadline, comment on our project tasks. Find the project on this link [12].

We used Google drive to create and update our presentations and other documents related to the project. As well, Github for uploading our code, creating and updating other issues and tasks. Find the project on this url [13].

VII. Scalability and Performance Benchmarking

The purpose of scalability benchmarking was to:

1. Find out how many concurrent connections per client will my single Node.js instance handle.
2. Find out how does the load capacity increase if more of my instances are added to the cluster.
3. Find out the optimal point of performance. How does a balanced and efficient mode of operation look like?
4. Ensuring optimal usage of resources (like: hard disk usage, CPU usage and RAM usage).
5. Which application design offers best service stability when memory, CPU, network limits on my instances are reached or single instances die abruptly in your cluster?

High Availability

In order to build even more highly performed applications, should be taken into consideration also:

- Better performed Load Balancer - DNS Config and Multiple LB to avoid single point of failure.
- Multiple Physical Machine (with LB on a third machine)
- Multiple Application Instances (Docker), this adapted to the needs of the application. Via the Load Balancer can be controlled the deletion or adding of nodes.
- Database (Multi-node)

Tools Used for Benchmarking

Apache JMeter: Apache JMeter™ is open source software written completely in Java. We used JMeter to load test functional behavior and measure performance [14].

Benchmarking Framework Goals

Here were the goals of benchmarking framework:

- Determining the number of concurrent requests, a single instance can handle.
- Finding service degradation and bottlenecks
  - o Service degradation can be because of I/O, CPU or Memory.
  - o Solution: monitoring the resource usage on instances.
- Finding load capacity increase or decrease if a single node is added/removed

Determining balanced and efficient mode of operation.

VIII. System Monitoring

During the tests it is needed to monitor resources such that is known where could be a bottleneck. For example, it can be monitoring network bandwidth, disk usage, CPU usage and memory usage.

As our environment is based on docker containers, we were in a need to monitor software for our docker instances.

We found an open source project [15] that can perform this operations..

CAdvisor can monitor and generate the graphs for all container physical resources.

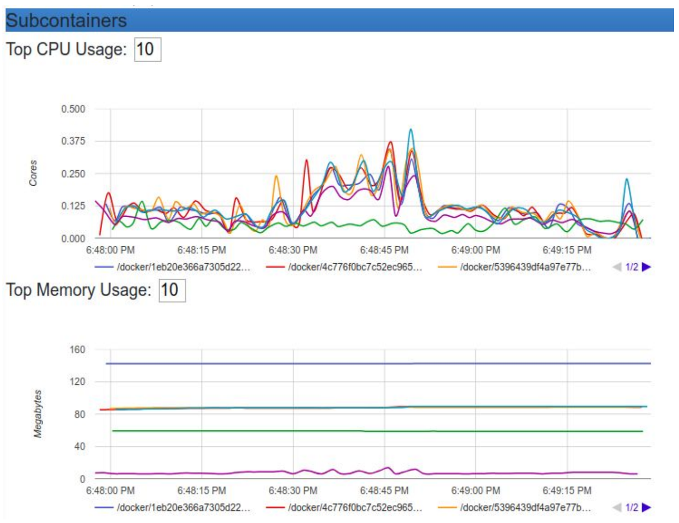


Fig. 6. CAdvisor graphs showing resource usage report

Benchmarking Configurations

We setup Apache JMeter to send two types of requests:

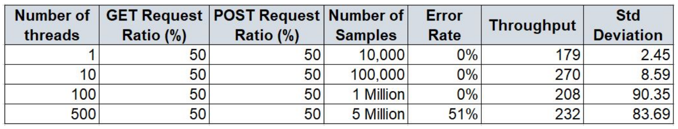
1. Create Reviews (Post)
2. List Reviews (Get)

JMeter shows following parameters in the summary of the tests:

1. Sample - number of requests sent
2. Avg - an Arithmetic mean for all responses (sum of all times / count)
3. Minimal response time (ms)
4. Maximum response time (ms)
5. Deviation - see Standard Deviation article
6. Error rate - percentage of failed tests
7. Throughput - how many requests per second does your server handle. Larger is better.
8. KB/Sec - self explanatory
9. Avg. Bytes - average response size

In addition, the initial testing was performed on following parameters:

The List Review request only fetches a limited number of records which we need to specify in the request parameters. We have used 10 records. This was required because if we fetch all the reviews in at the same time then the response rendering time depends upon the size of the results. If there are more results the response time would be higher and if there are less results then the response time would be higher.



Number of threads	GET Request Ratio (%)	POST Request Ratio (%)	Number of Samples	Error Rate	Throughput	Std Deviation
1	50	50	10,000	0%	179	2.45
10	50	50	100,000	0%	270	8.59
100	50	50	1 Million	0%	208	90.35
500	50	50	5 Million	51%	232	83.69

Fig. 7. Level one benchmarking results

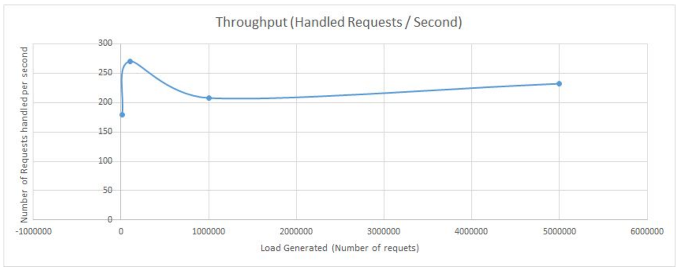


Fig. 8. Level one benchmarking results graph

With a sample of 2756783, around 50.19% of requests failed because of Response code: Non HTTP response code: java.net.ConnectException Response message: Non HTTP response message: Connection refused: connect. Response headers: HTTPSampleResult fields: ContentType: DataEncoding: null From error details it seems that the bottleneck is happening from JMeter end java.net.ConnectException: Connection refused: connect
It took over a night to execute 500 Threads and 10000 loop count. Infact JMeter was behaving too slow and finally it it crashed because of Uncaught Exception java.lang.OutOfMemoryError: Java heap space. See log file for details.

Initially, the Out of memory error started coming but it continued performing tests and now its completely dead. Unfortunately, it does not complete things in minutes. It takes long time.Initially things were failing but it was executing the tests. But after a long number of samples it completely stopped working. For smaller number of threads it works. For real huge data it does not. I still did not did any testing for over a million requests.So it was expected, a single machine cannot handle this load. For that thing I hope it would completely not work. At least I did not expect this. On server machines every MB of RAM causes extra dollars and we have Core i5 reserved for only this task.

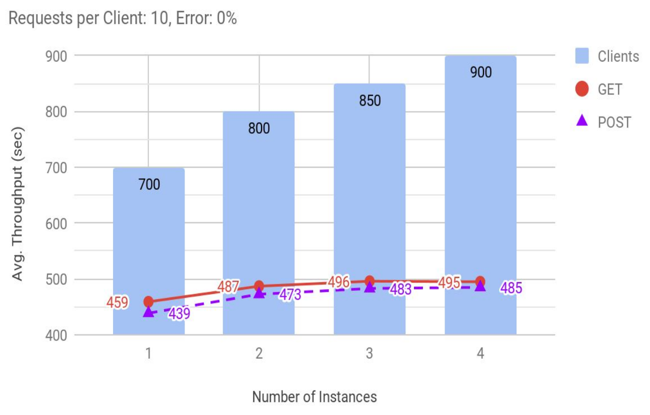


Fig. 9. Level one benchmarking results

This line graph explains the trend of throughput of single machine considering multiple setups. We created the scenarios of 1, 2, 3, 4 instances with respectively 700, 800, 850, 900 Clients connections and with 10 simultaneous GET and POST requests per Client. In the graph is shown the average throughput for different setups in the ideal cases with 0% Error rate of requests sent. If we would increase the number of Clients connected and requests per client the Error rate % would increase. This is the reason why we came up with this number of Clients connections and requests per client.

So, firstly on the single machine it was tested one instance with 700 Clients and 10 GET and POST requests per client. The average throughput performed with 0% Error was 459 GET requests sent and 439 POST requests sent.
Secondly, were tested two instances with 800 Clients and 10 GET and POST requests per Client. The average throughput was 487 GET requests sent and 473 POST requests sent.
Thirdly, were tested three instances with 850 Clients and 10 GET and POST requests per Client. The average throughput was 496 GET requests sent and 483 POST requests sent.
Lastly, were tested four instances with 900 Clients and 10 GET and POST requests per Client. The average throughput was 495 GET requests sent and 485 POST requests sent.

It is noticed that after two instances, if we keep increasing the number of instances per single machine, the performance stays almost constant, does not get improved. From one instance to three instances the performance is increased by 21%.

850 Clients - 700 Clients = 150 Clients
150 Clients / 700 Clients = 0.21
*0.21 * 100 = 21%*

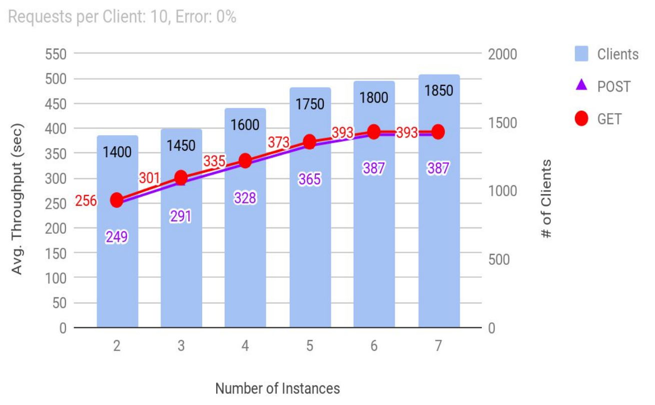


Fig. 10. Multiple machines throughput

This line graph explains the trend of throughput of multiple machines considering multiple setups.
We created the scenarios of 2, 3, 4, 5, 6, 7 instances with respectively 1400, 1450, 1600, 1750, 1800, 1850 Clients connections and with 10 simultaneous GET and POST requests per Client.
In the graph is shown the average throughput for different setups. The throughput showed 0% Error rate of requests sent until the scenarios with 6 instances. When number of instances was increased to 7 the Error rate % increased and the number of throughput stayed constant.

1. On the two machine it were tested two instances with 1400 Clients and 10 GET and POST requests per client. The average throughput performed with 0% Error was 256 GET requests and 249 POST requests sent.
2. Tested three instances with 1450 Clients and 10 GET and POST requests per Client. The average throughput was 301 GET requests and 291 POST requests sent.
3. Thirdly, were tested four instances with 1600 Clients and 10 GET and POST requests per Client. The average throughput was 335 GET requests and 328 POST requests sent.
4. Next, were tested five instances with 1750 Clients and 10 GET and POST requests per Client. The average throughput was 373 GET requests and 365 POST requests sent.
5. Then, were tested 6 instances with 1800 Clients and 10 GET and POST requests per Client. The average throughput was 393 GET requests and 387 POST requests sent.

6. Lastly, we tested seven instances with 1850 Clients and 10 GET and POST requests per Client. The average throughput was 393 GET requests and 387 POST requests sent with more than 0.01% Error rate.

It is noticed that after six instances, if we keep increasing the number of instances per two machines, the performance stays almost constant, does not get improved.
From two instance to six instances the performance is increased by 29%.

1800 Clients - 1400 Clients = 400 Clients
400 Clients / 1400 Clients = 0.29
*0.29 * 100 = 29%*

These results show that HTTP Get and Post Requests are performing almost same time. As the POST Request does not do any considerable computations. Also increasing the number of nodes on a single machine does not ensure high scalability. As the instances share the same host and docker container needs to manage instances which is also an overhead.

The experiment shows that increasing number of nodes from 1 to 2 or 2 to 4 raised the throughput. But increasing the number of nodes from 4 to 8 does not result to a better performance. For example, using only one instance the throughput was 501, but increasing docker instances to 4, reduced the throughput. Increasing the instances number to 8 increased the throughput, but it did not doubled the throughput. This means that docker instances should be added when needed

IX. Evaluation

Bottlenecks

- Network
- Application Server
- Database Scalability (Multi-node Database)
- Load Balancing algorithm - Least Connections / Round Robin

Best Practices

1. Always use proper network infrastructure
2. HAProxy vs Nginx
3. Jmeter CLI
4. Test Environment on Seperate machine
5. cAdvisor for Docker Stats
6. Run Multiple Jmeter to reduce the weird pausing

X. Future Architecture

We propose a better architecture based on our experiences.

1. There must be multiple load balancers. Such that if one load balancer crashes then the other one can resume the workload.
2. A load balancer is connected to every node and it uses some intelligent algorithm to identify least busy node and directs the current request to that node.
3. There are multiple database instances and each database is consistent.

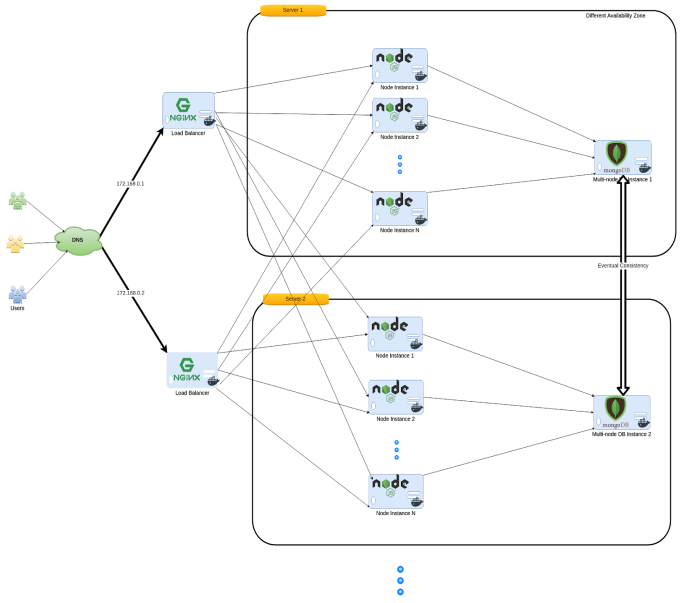


Fig. 11. An system architecture that consists of multiple nodes on multiple physical machines.

XI. Conclusions

We saw that an efficient mode of work depends on multiple components in an architecture. For example it can be location of host machines, capacity of design of scaling up and out dynamically, and also updated technologies like container solution can strongly support your application. For our rating application we came up with a highly scalable application design that can be used in your local cloud and for millions of users. Throughout the evaluation of multiple designs, it gave us a lot of new insights of enterprise application development.

Lessons Learnt

1. Building RESTful API with Node.js and Express

2. How to implement a load balancer in an application architecture
3. Nginx as a reverse proxy and Load balancer algorithms
4. Dockerizing a Node.js application
5. Building a scalable application
6. Testing an application scalability in different scenarios.
7. How to manage in group a project similar to this scenario.

References

[1] Stefan Tai, David Bermbach and Erik Wittern, “Cloud Service Benchmarking”, Springer International Publishing AG 2017, pp. 21, in press.

[2] Ludmila Cherkasova, “FLEX: Load Balancing and Management Strategy for Scalable Web Hosting Service”, Hewlett-Packard Labs, 1501 Page Mill Road, Palo Alto, CA 94303, USA, 2000, pp. 2in press.

[3] Roy T. Fielding, and Richard N. Taylor, “Principled Design of the Modern web Architecture”, Irvine, 2002, in press.

[4] Derek DeJonghe, “The complete NGINX cookbook”, First Edition, O’Reilly, March 2017, 1.3 Load Balancing methods, pp. 5, in press.

[5] Wikipedia contributors. (2018, May 16). Leader election. In *Wikipedia, The Free Encyclopedia*. Retrieved 18:57, July 29, 2018, from https://en.wikipedia.org/w/index.php?title=Leader_election&oldid=841536260

Web References

[6] Adnan Rahić, RESTful API design with Node.js, hackernoon.com, Mar 5, 2017 https://hackernoon.com/restful-api-design-with-node-js-26ccf66eab09

[7] Chris Sevilleja, Build a RESTful API Using Node and Express 4, scotch.io, April 15, 2014 https://scotch.io/tutorials/build-a-restful-api-using-node-and-express-4

[8] Wikipedia contributors. (2018, July 18). Apache Cassandra. In *Wikipedia, The Free Encyclopedia*. Retrieved 20:05, July 29, 2018, from https://en.wikipedia.org/w/index.php?title=Apache_Cassandra&oldid=850826292

[9] Wikipedia contributors. (2018, July 26). MongoDB. In *Wikipedia, The Free Encyclopedia*. Retrieved 20:07, July 29, 2018, from https://en.wikipedia.org/w/index.php?title=MongoDB&oldid=852052919

[10] Anand Mani Sankar, A sample Docker workflow with Nginx, Node.js and Redis, anandmanisankar.com, March 30, 2015 http://anandmanisankar.com/posts/docker-container-nginx-node-redis-example/

[11] Aaron Alexander**,** Load Balancing with Nginx and Docker, sep.com, February 28, 2017 https://www.sep.com/sep-blog/2017/02/28/load-balancing-with-nginx-and-docker/

[12] NodeJS Scalable Application Designs, D.R., N.K., U.N. asana.com, May 20, 2018, https://app.asana.com/0/687800635596872/687800635596872

[13] naveedkamran/nodejsapp, naveedkamran, RucajDenisa, umar-nawaz, github.com, April 29, 2018, https://github.com/naveedkamran/nodejsapp

[14] Apache JMeter, https://jmeter.apache.org

[15] CAdvisor https://github.com/google/cadvisor