International Journal of Innovative Computing, Information and Control
Volume **17**, Number **6**, December **2021**

ICIC International ©2021 ISSN 1349-4198
pp. 1937–1953

ENHANCING HEALTH MONITORING SYSTEM PERFORMANCE WITH GEO-IP DNS BASED LOAD BALANCING

Nico Surantha, Christopher Alvin and Theresa Lusiana

Computer Science Department, BINUS Graduate Program – Master of Computer Science
Bina Nusantara University
Jl. K. H. Syahdan No. 9, Kemanggisan, Palmerah, Jakarta 11480, Indonesia
nsurantha@binus.edu; { christopher.alvin001; theresa.lusiana }@binus.ac.id

Received June 2021; revised September 2021

Abstract. *Health monitoring systems demand a huge data transaction, especially if it was set on a multi-regional architecture. With a huge data transaction, there is a higher probability of overloaded number of requests. Conventional load balancer requires tunnelling, which means it takes longer time to reach the designated system that handles request and go back to the request source. This paper proposes the enhancement of the current multi-regional architecture in health monitoring systems by applying Domain Name System (DNS) load balancing as the better alternative to conventional load balancer. The proposed architecture significantly increases the throughput and decreases the response time. The proposed architecture performance is compared with the traditional load balancing with a weighted round robin and another sleep monitoring system architecture with Kubernetes but without load balancing. When receiving the request, the data center has to allocate these requests efficiently so that the response time should be minimized to avoid overloading or congestions. Response time, throughput, completion time, Central Processing Unit (CPU) usage and error rate are the metrics that we use here. The proposed architecture achieves the lowest average response time, the highest average throughput, the lowest error rate and the lowest completion time for 50,000 requests hit from each region (Oregon and Singapore).*

Keywords: Domain Name System (DNS) load balancing, Response time, Cloud computing, Throughput, Health monitoring system

1. Introduction.

Clinical home testing is currently targeting sleep disordered breathing, and the data supporting the use of home sleep apnea devices has been reviewed recently [1, 2]. The personal wellness goal of sleep-monitoring in order to optimize health also stands to be achieved through longitudinal monitoring and self-tracking [3]. Sleep monitoring system is a system to track the sleep cycle of humans. It requires high sensitivity (ability to detect true sleep) and accuracy (overall ability to detect true wake and sleep). They perform 24/7, generating overwhelmingly large datasets (Big Data), with the potential of offering an unprecedented window on users' health [4].

With the use of cloud-based applications, it is easier for users on the web to make use of the widely available services and resources [5]. Cloud computing enables computing over the Internet performs at high agility and high efficiency [6]. Cloud computing provides its clients with a virtualized network access to applications and services [7]. It provides service to customers of various requirements with the aid of online computing resources [8]. A cloud computing model is efficient if its resources are utilized in the best possible way and such efficient utilization can be achieved by employing and maintaining proper management of cloud resources and powerful resource scalability techniques [8]. Imbalance

DOI: 10.24507/ijicic.17.06.1937

traffic often occurs, causing some servers having excess requests and other servers having underwhelming amounts of requests. This can render cloud computing left with unbalanced machines which have a huge gradient of user tasks and resource utilization [9].

With the rise of the Internet of Everything (IoE), the number of smartphones and devices connected to the Internet increases. This causes a huge rise in large-scale data, that causes many problems such as bandwidth load, slow response speed in traditional cloud computing models [10], high latency, and low Spectral Efficiency (SE) [11]. Hence, edge computing, a new computer paradigm, emerges. It emphasizes on getting closer to the user and closer to the sources of the data [10]. It carries out a substantial amount of computation, storage, communication tasks, offering for low latency, energy efficient, and agile computation augmenting services. Edge computing is more suitable to be integrated with Internet of Things (IoT) to provide efficient and secure services for a large number of end-users, and edge computing-based architecture can be considered for the future IoT infrastructure [12].

This is where an effective load balancing algorithm plays a crucial role. This load balancing distributes the dynamic workload evenly among all the nodes [13]. The efficiency of task allocation to the cloud determines the effectiveness of the load balancing algorithm [14]. A Domain Name System (DNS) load balancer allows application-level feedback and health monitoring by means of a load metric used to decide which Internet Protocol (IP) addresses to present when referencing the alias name. There are two other definite benefits of DNS load balancers, which are balancing the load without modifying the traffic patterns or adding network gateways that may become availability and performance bottlenecks and supporting transparently all protocols [15, 16]. On an additional note, it supports all protocols transparently.

Our research objective is to develop a more efficient health monitoring system with DNS load balancer for sleep monitoring systems. By using DNS load balancing, we hope to decrease the response time so each request is handled faster. As requests are handled faster, for a given period of time, the system will be able to handle more requests compared to traditional load balancers. With this approach, we can increase throughput, decrease error rate, decrease response time and lower completion time compared to the previous work presented in our literature review.

We compare our sleep monitoring system with proxy and without load balancers. The load balancing method we propose is a combination of external load balancer using DNS load balancing and internal load balancer using round robin algorithm. We use some metrics for comparison including throughput, percentage of request success, error rate, completion time and response time.

Our incremental contributions in this research are

1) Geo-IP DNS Load Balancing that can reduce response time, reduce error rate and test completion times;
2) Multi Region/Geographical High Availability if one region is down, still having another to backup the request.

Based on our research, our proposed architecture achieves the lowest average response time, the highest average throughput, the lowest error rate and the lowest completion time for 50,000 requests hit from each region (Oregon and Singapore).

The remaining of this paper is arranged as follows. We present our literature review in Section 2. We present our proposed architecture in Section 3. After that, we have compared and shown the results of Section 4. Finally, we conclude the paper and expose possible areas of improvement and our future work regarding load balancing algorithms in healthcare in Section 5.

2. **Literature Review.**

[17] proposed an IoT-based sleep quality monitoring system, which comprises three main components: sensor gateway to gather the data, an IoT platform to store and process data from the sensors and lastly the web-based dashboard designed for users and/or health practitioners to see the result. Firstly, the IoT platform implements two architectures, which are microservice and event-driven. The microservice divides the system into five services: sensor-gateway, sensor-data-persister, sleep stage classification, sleep quality quantification and web application gateway, which can be developed independently, with minimal coupling. The evaluations were performed on a single machine with 4 Central Processing Unit (CPU) with a clock rate of 2 gigahertz (GHz) and 15 gigabyte (GB) Random Access Memory (RAM). The positive results gathered from this paper are 55.85 per cent decrease in response time and 34.76 per cent increase in throughput compared to traditional methods. However, the rate of increase in memory usage per instance replication is lower than traditional methods.

[18] simulates the impact of container orchestration on the sleep monitoring system. The proposed architecture [18] wants every service in the sleep monitoring system to be containerized inside Docker containers and managed by container orchestrations. The results show that Kubernetes can achieve the lowest CPU Utilization, error rate, and also network bandwidth compared to the other container orchestration such as Docker Swarm and Nomad. This method can also improve scalability from the applications, so that configuration and maintenance can be done easily.

[19] proposed a load-balancing method with a proposed decentralized architecture which periodically detects short-term overload hiccups and autonomously handles them using geographical load balancing to reduce the risk of decreasing performance. The architecture dictates that each geographical agent comprises the monitoring module, which constantly monitors the incoming requests and the status of the available resources to detect application overload; the communication module, which is in charge of broadcasting its status to other agents and receiving other agents' statuses; and the control module that quickly adapts the application/service to the detected overload events. The results are based on the prototype evaluated on Amazon Web Services' Europe, US and Asia data centers. The positive outcome of this is the great increase in the number of requests during overload periods while still maintaining an acceptable quality of service.

[20] proposed a cloud load balancing mechanism that can be applied to both virtual web servers and physical servers. The findings demonstrate that when many users log in at the same time, cloud service performance, based on the architecture proposed in this study, will balance loading performance. Cloud load balancing considers processing power and load, so it can minimize the chance that a server cannot handle any request.

[21] proposed a load balancing algorithm with graph colouring based on genetic algorithm. The proposed method is compared with three other techniques, which are Cloud First, Round Robin, and Priority. The result of the experiment shows that the proposed technique generates the least network traffic, and thus, the probability of network failures is lower than with the other three techniques compared with the proposed method.

[22] compares load balancing algorithm between link load-balancing and server load-balancing scenario. The difference is that a link load-balancer will receive request from different clients and distributes that over intermediate nodes before going to server. On the other hand, server load-balancer will distribute request to various servers. Packet received ratio is better with server load-balancer than that with link load-balancer.

In the research papers we reviewed, there is room for improvement for optimizing load balancing in multi regional scale. Hence, our paper is created in the hope to fill the research gap in the previous research that is not optimized to scale to multi regional use cases with Geo-IP DNS load balancing. To view the current state of the art of load balancing

Table 1. Summary of the literature review of this paper

Paper	Proposed method	Result
[17] in *High-Performance and Resource-Efficient IoT-Based Sleep Monitoring System*	An IoT platform architecture designed based on the microservices and event-driven architecture.	Response time decreases by 55.85%, throughput increases by 34.76%, the rate of increase in memory usage per instance replication is lower.
[18] in *Sleep Quality Monitoring System Based on Container Orchestration*	A system architecture for end-to-end sleep monitoring systems by using Kubernetes as container orchestration.	Good result in availability, flexibility and scalability.
[19] in *Mitigating Impact of Short-Term Overload on Multi-Cloud Web Applications through Geographical Load Balancing*	A decentralized system that timely detects short-term overload situations and autonomously handles them using geographical load balancing and admission control.	Acceptable quality of service, greatly increase the number of requests served during overloading periods, develop a more accurate overload detector.
[20] in *CLB: A Novel Load Balancing Architecture and Algorithm for Cloud Services*	Cloud load balancing with weighted round robin load balance mechanism.	Unable to handle excessive computational requirements, can balance the loading performance when users logged in at the same time.
[21] in *A Load Balancing Algorithm for Mobile Devices in Edge Cloud Computing Environments*	Load balancing algorithm with graph colouring based on genetic algorithm.	Generate the least network traffic, and thus, the probability of network failures is lower than with the other three techniques compared with the proposed method.
Our architecture	Geo-IP DNS based load balancing.	Greatly increase throughput, greatly decrease response time.

architecture we reviewed, please refer to Table 1. The table shows a comparison of existing health monitoring without load improved along with high availability. The architecture is explained in Section 3.

3. Methodology/Proposed Design.

In this section, we will firstly explain the two architectures that we want to compare with our proposed architecture. The two architectures will be explained in Sections 3.1 and 3.2 respectively. Next, we will explain our proposed design from the general architecture to the configuration for the Kubernetes container orchestration as well. The general architecture that we propose is explained in 3.3. The general architecture will consist of the explanation for the cloud configuration as well as how DNS load balancing works. In Section 3.4, we will explain the configuration for the Kubernetes container orchestration that is used in this research.

3.1. Conventional Kubernetes without load balancing.

The conventional Kubernetes without load balancing has 2 masters and 4 workers station in one region. The internal load balancer (ingress) implements round robin algorithm. During our test, they are put in Singapore server.

3.2. **Weighted round robin load balancing.** The weighted round robin load balancing has 2 machines in one region and 2 machines in another, implementing weighted round robin load balancer. During our test, 2 machines are put in Singapore, and 2 machines are put in Oregon, California.

3.3. **Proposed Geo-IP DNS load balancing.** We are enhancing from the previous model that we reviewed [18], where there are 6 services: Classification, Scheduler, Quantification, Apache Kafka, Data Persister, Web Socket, represented as Service 1 and 2 non-exhaustively in Figure 1. Healthcare monitoring system is best if it can be applied at a global scale so that every branch in the world can access the system. However, the problem with the previous cloud computing model we reviewed is that it has a bottleneck in the response time and the error rate. Our proposed architecture will add a mechanism to support a multi-regional operation for the model, without compensating on the performance. It is a simple model to achieve reliable scalability of computing services in a cloud environment with a relatively large number of services. With the addition of two load balancers: the regional load balancer (the DNS load balancer) and the internal load balancer (which is embedded in the ingress in each region), we are hoping to reduce this bottleneck. The regional load balancing algorithm is based on the proximity of the IP address from the user to the region. The internal load balancing (inside ingress) algorithm will be based on round robin. We will delve into the general scenario in Figure 1 in the next paragraph.

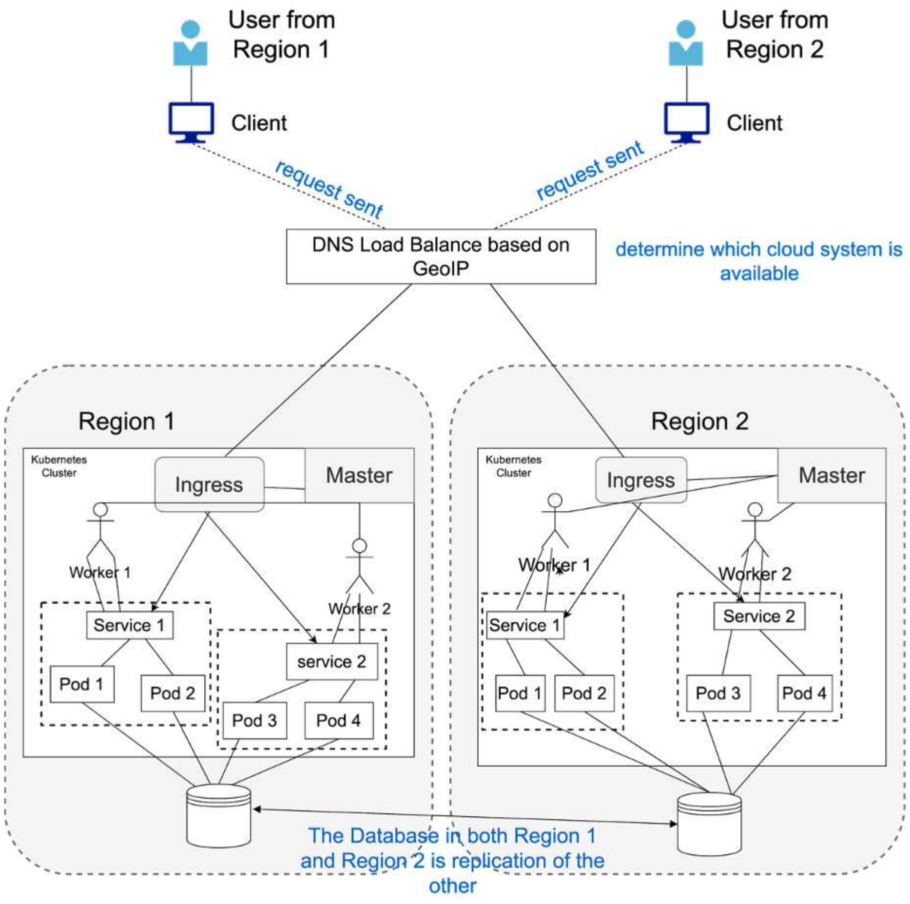

Figure 1. The diagram of the general version of our proposed DNS load balancing architecture

A generalized version of our proposed architecture in Figure 1 will be explained here now. First, one user from region 1 will create a request. The request will then pass through the regional DNS load balancer. The DNS load balancer will answer the specific request by replying to it with an IP address for the request to go. If Region 1 is not available for request handling, the request will be passed to Region 2, which is the next nearest region. Upon arrival of the request in the specific system that will handle it, it will go through ingress, where there is an internal load balancer to assign by which worker the request will be handled.

In Figure 2, we simulate how the DNS load balancing works. Firstly, a computer X asks example.com for record from Authoritative Root Name Server. The request is then passed to the DNS load balancer. When the request arrives, the DNS load balancer checks the Geo-IP Database for the source location of the IP address. The database then returns an IP address record belonging to the specific country. Then, the last step is to return example.com IP address of the nearest available server that is able to handle the request. A null case is when the country record is not found in the database. Then, if the country is not found, it will return a random example.com IP address that can handle the request.

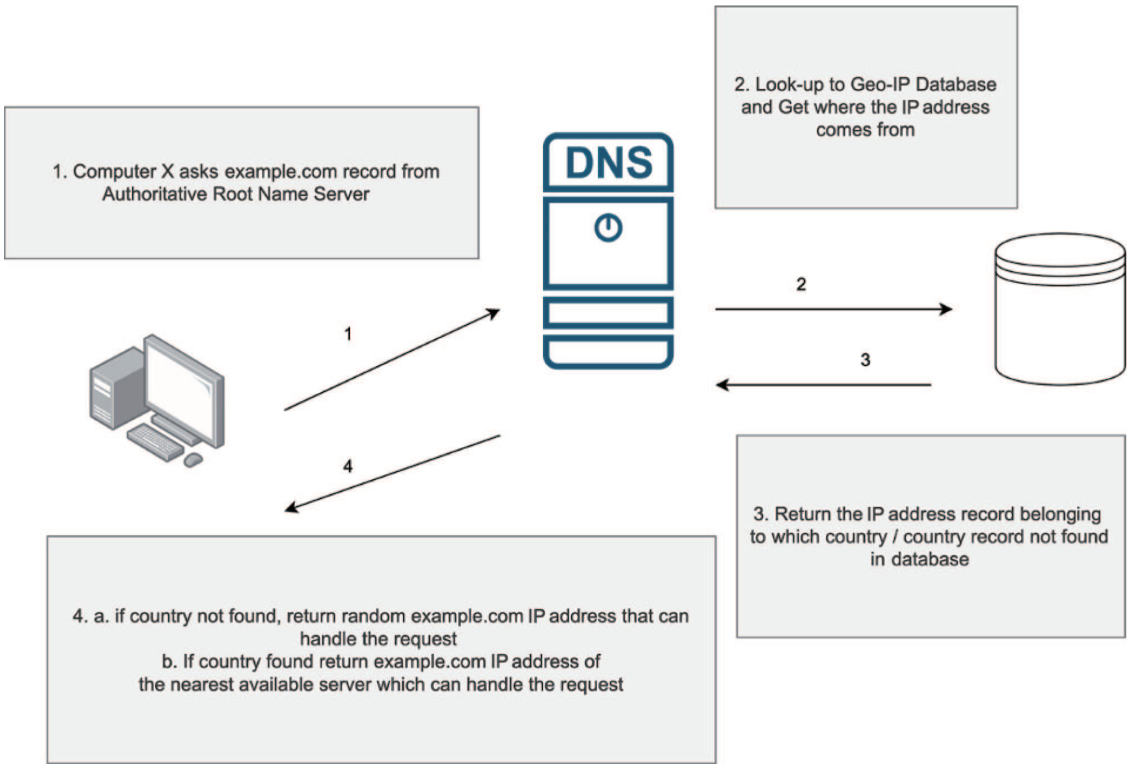
1. Computer X asks example.com record from Authoritative Root Name Server
DNS
2. Look-up to Geo-IP Database and Get where the IP address comes from
1
2
3
4
3. Return the IP address record belonging to which country / country record not found in database
4. a. if country not found, return random example.com IP address that can handle the request
b. If country found return example.com IP address of the nearest available server which can handle the request

FIGURE 2. The diagram of the DNS load balancing mechanism

Figure 3 shows a simple flow chart explaining the round robin algorithm used in the internal load balancer. This algorithm starts with the incoming request to the ingress. The request will be passed to the first pod. The request is then handled by the first pod. Once it is done being handled, the next request in queue will be considered to be handled by the next pod. This process is iterative until it has reached the last pod. Upon reaching the last pod, the next request will be handled by the first pod.

3.4. **System specification.** [20]'s architecture has four nodes, two for each region, Oregon and Singapore. It does not use Kubernetes. [18]'s architecture has only one master node located in Singapore, and four worker nodes in Singapore. In Table 2, we display the

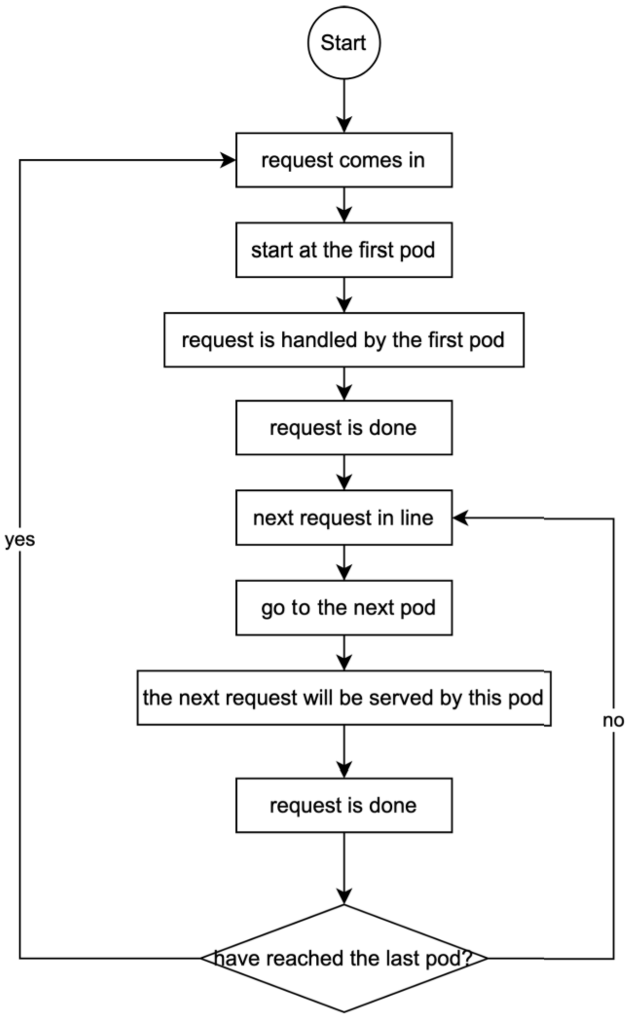

FIGURE 3. The algorithm of the round robin

TABLE 2. System specification used for this research

Configuration	Value
Cloud Platform	Google Cloud Platform - N2-Standard-2
Operating System (OS)	Container Optimized Linux
CPU	2 vCPU
RAM	8 GB
Disk	25 GB
Master Nodes	2
Worker Nodes	4

system specification used for the purpose of this research. For our proposed architecture, we set up two master nodes each for the different regions, Oregon and Singapore. Hence, each region needs its dedicated master node. We set up four worker nodes, and placed them two at each region. We used the smallest CPU N2 type instances and RAM, which are 2 vCPU and 8 GB consecutively.

4. **Result & Discussion.**

In this section, we discuss our experiments and results. For evaluation, we compare our proposed architecture with that in [18] and [20] architecture.

We compare the three architectures in two cases: 50,000 and 100,000 concurrent requests. We decided to limit the experiments with 50,000 and 100,000 concurrent requests for the purpose of faster testing time. Also, these tests are quite representative and scalable to higher number of requests too.

4.1. **Evaluation scenario.** The evaluation objective is to compare the performance of our proposed architecture to two other architectures, so that we would know which performs best. We would cover and explain the metrics that we choose to evaluate the architectures in 4.1.2. We present the test bed of our experiment in 4.1.3. Lastly, in Section 4.1.4, we are explaining the scenarios that we test in this experiment.

4.1.1. *Evaluation objective.* Our proposed architecture is meant to decentralize the information exchange system. Our experimentation is only limited to the information exchange, not considering the machine learning behind it.

4.1.2. *Metrics.* Below are the metrics that we use to gauge the performance of our proposed architecture compared to the two other architectures.

a) Completion time
The time needed to complete all the incoming requests.

b) Response time
The time needed for the request to return back to the application.

c) Throughput
The amount of requests attended over the time of testing.

d) CPU usage
A gauge on how effective your testing is. If the CPU utilization reaches 100%, it can no longer process requests and the throughput will eventually flatten. This is measured by averaging all CPU usage in all of our worker nodes.

e) Error rate
Formula is

$$\textit{error rate} = \frac{\textit{number of failures to serve request}}{\textit{total number of incoming requests}} \tag{1}$$

4.1.3. *Test bed.* We will use two regions for the testing scenario, which are Oregon (in California) and Singapore. These two regions indicate where the data comes from. Figure 4 displays the testing scenario that we are using for this research, displaying that requests can come from Singapore or Oregon. There are two cloud systems that are working to handle requests, one is located in Singapore and the other one is located in Oregon. The database for each system (as seen on the bottom part of the image) is a replication of the other. When a user in Oregon comes with a request, it passes the DNS load balancer, which identifies which the nearest cloud system is available to handle the request.

4.1.4. *Scenario.* Data that we are collecting from sleep monitoring system is electrocardiogram (ECG) signal from the interpreted cardiac activities from a human. Our scenario is compared to two other architectures, which are presented in [20] and [18] as shown in Table 3 where we label each method with a number as a reference for Table 4. 50,000 requests are defined as 50,000 incoming requests from Singapore and 50,000 incoming requests from Oregon (in California). The same goes for 100,000 requests, meaning 100,000 incoming requests from Singapore and 100,000 incoming requests from Oregon (in California). For each scenario, we run them 30 times and plot them in a cumulative distribution function, displayed in Section 4.2.

Total concurrent requests stated in Table 4 is defined as the number of incoming data in a given unit time. For example, in the health monitoring case, there are 50,000 heartbeat

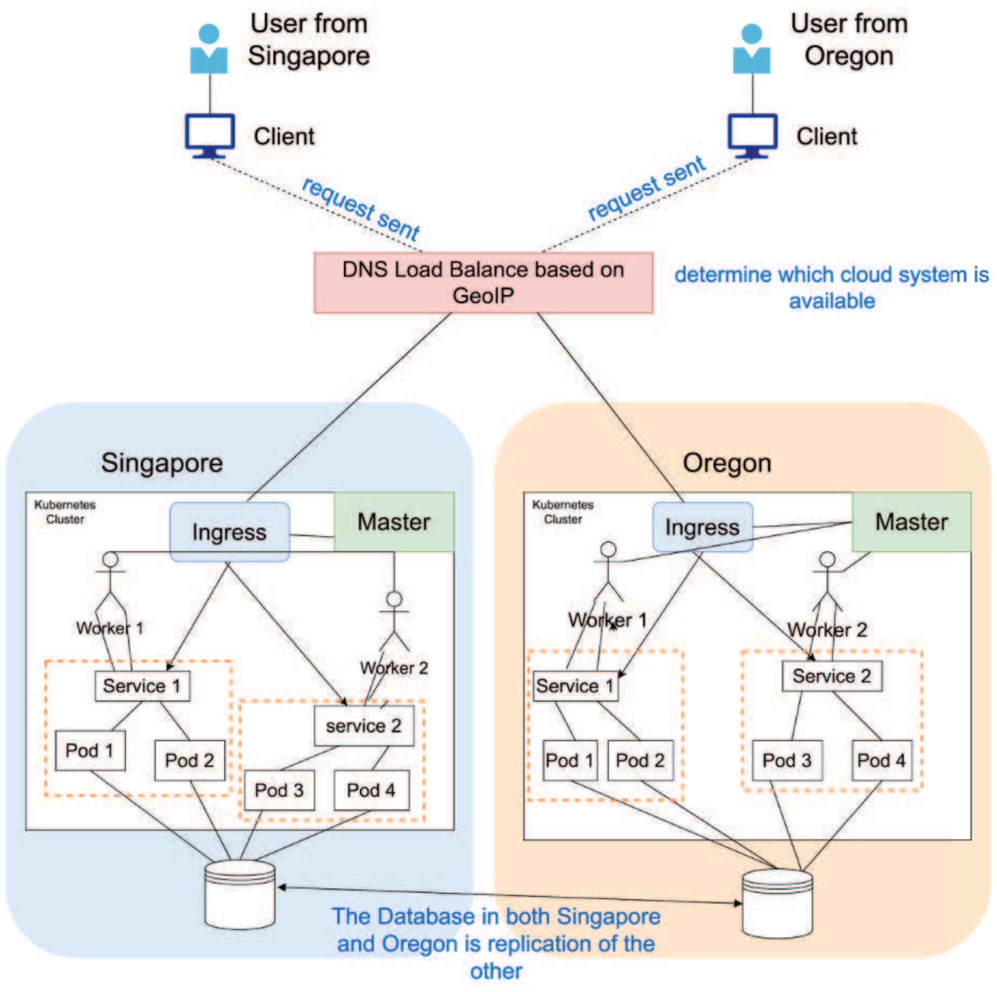

FIGURE 4. The diagram of the experimentation version of our proposed DNS load balancing architecture

TABLE 3. The methods compared in this research with their respective numbered method

No	Method	Reference
1	Geo-IP DNS Load Balancing	Our method
2	Kubernetes without load balancing (uni-location in Singapore)	[18]
3	Weighted Round-Robin Cloud Load Balancing (Load Balancer located in Singapore)	[20]

TABLE 4. All the scenario combinations that will be presented in the research result

Scenario	Method	Total concurrent requests
1	1	50,000
2	1	100,000
3	2	50,000
4	2	100,000
5	3	50,000

records coming into the database. If the person is asleep for every second: 50,000 is divided by 3600 (60 minutes times 60 seconds per minute), resulting in approximately 14 hours of sleeping. Assuming a normal person would sleep for 7 hours, this means that 50,000 heartbeat records equal to the recorded heartbeats of 2 persons sleeping for 7 hours each. Table 4 shows the scenarios that we are using in this research, along with which method we are using in that scenario and the number of total concurrent requests.

4.2. **Experiment's result.** In this section, we will present the result of our experiment in terms of the completion time, response time, throughput, CPU usage and error rate.

We would like to level the playing field by stating that for the three architectures compared in 4.2 here, all three architectures use four nodes, but with different architecture configurations. Our proposed architecture uses four worker nodes in total, with two in Singapore and the other two in Oregon. [18]'s architecture has only one master node in Singapore, but still has four worker nodes, all in Singapore. [20]'s architecture has four virtual machines in total, two of which are located in Singapore and the other two are located in Oregon. The only difference between our proposed architecture and [20]'s architecture is the load balancer, our proposed architecture uses DNS load balancer, and meanwhile, [20]'s uses weighted round robin load balancer.

For the compared throughput in Figure 5, we cut off the maximum response time at 2,000 milliseconds. We present the result in cumulative distribution function. As you can see in Figure 5, the y axis is in % of total run, which refers to the percentage of test runs out of the total 30 test runs with 50,000 or 100,000 data per test run. The x axis represents the throughput in request/second.

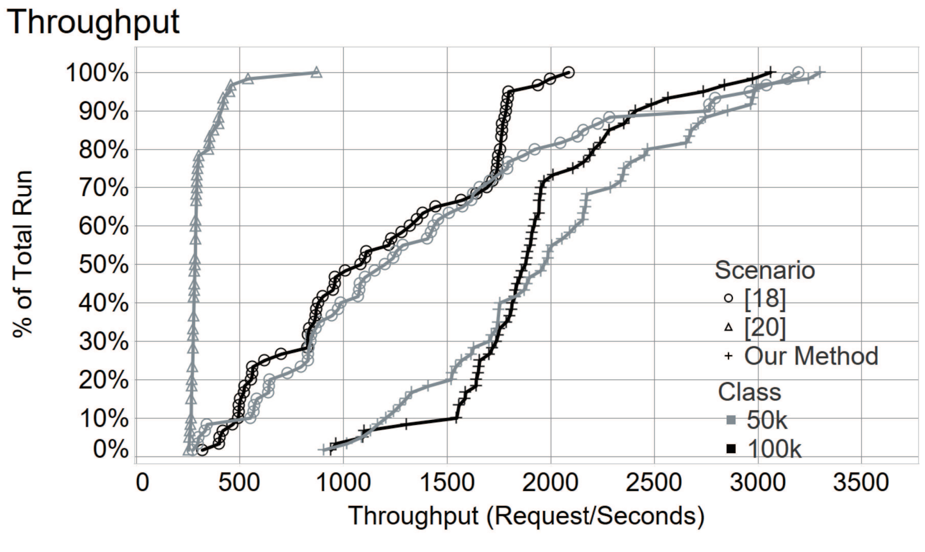

FIGURE 5. The comparison for throughput for scenarios 1 to 5

From the graph in Figure 5, the two scenarios, displayed on the bottom right part of the graph, perform better than the rest of the three scenarios. We can see that generally, our proposed architecture (scenarios 1 and 2) performs best, followed by [18]'s architecture (scenarios 3 and 4) and finally [20]'s architecture. From the displayed comparison in Table 5, the average throughputs for our method are the highest at 2088.8 requests per seconds and 1912.7 requests per seconds. This is followed by [18]'s architecture with 1377.1 and 1158.5 requests per seconds for scenarios 3 and 4. Lastly, [20]'s architecture has the lowest throughput of 315.5 requests per seconds (in scenario 5). Higher throughput in our

Table 5. The comparison table for throughput for the 5 scenarios

Throughput (Request/seconds)				
Scenario	Average	Max	Min	Std. Dev (Pop)
1	2088.8	3299.5	905.6	585.4
2	1912.7	3061.4	940.3	424.5
3	1377.1	3196.9	273.6	772.7
4	1158.5	2086.6	319.4	525.5
5	315.5	872.1	253.7	93.3

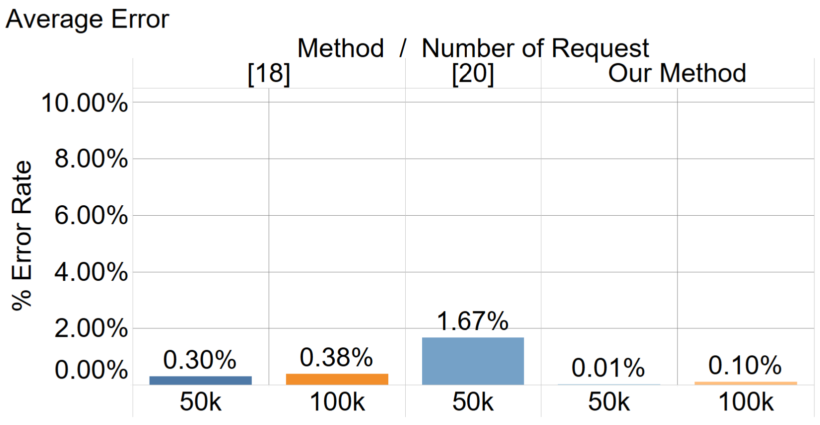

Figure 6. The comparison for error rate for scenarios 1 to 5

architecture is due to the faster response time in our architecture. With faster response time, our proposed architecture can handle more requests at the same, given amount of time than [18] and [20]. A detailed reason for the faster response time will be explained in the later part of the paper.

From Figure 6, the error rate is greatest for [20]'s architecture, followed by [18]'s architecture, and finally our method at the lowest. This implies that our method creates the least number of error requests. As seen in Table 6, our method has the lowest average error rate of 0.01% and 0.10% for scenarios 1 and 2 respectively. [18]'s architecture has an average error rate of 0.30% and 0.38% for scenarios 3 and 4 respectively, and finally [20]'s error rate is the highest at 1.67%. The Internet itself does not guarantee reliable packet delivery. The Internet is free to destroy packets if they are overloaded. The request goes to and comes back from the pod where the request is handled. This is the route that is taken by the request. The longer the route, the higher the chance of it gets destroyed before reaching the origin. [20]'s architecture requires an Oregon request to go to Singapore and comes back to Oregon again. This is a tediously long route for the request. Hence, the error rate is higher in [20]'s architecture, compared to that in our architecture. The error rate in our architecture is lower than that in [18]'s architecture because our architecture has two ingresses, meanwhile [18]'s architecture has only one, that is located in Singapore, even though both have the same number of worker nodes. Thus, for an Oregon request, it needs to come to Singapore to be handled. Meanwhile in our architecture, it can be handled in Oregon.

In Figure 7, we see that our method reaches almost 100% of total run after 40-100 seconds. Meanwhile, [20]'s and [18]'s architecture take longer to complete almost 100% of the total requests coming in. This implies that our proposed method is faster in completing the incoming request. As displayed in Table 7, our method has the shortest average

Table 6. The comparison table for error rate for the 5 scenarios

Error rate (Percentage)				
Scenario	Average	Max	Min	Std. Dev (Pop)
1	0.01%	0.28%	0%	0.04%
2	0.10%	2.05%	0%	0.39%
3	0.30%	2.80%	0%	0.58%
4	0.38%	1.68%	0%	0.41%
5	1.67%	7.82%	0.01%	1.87%

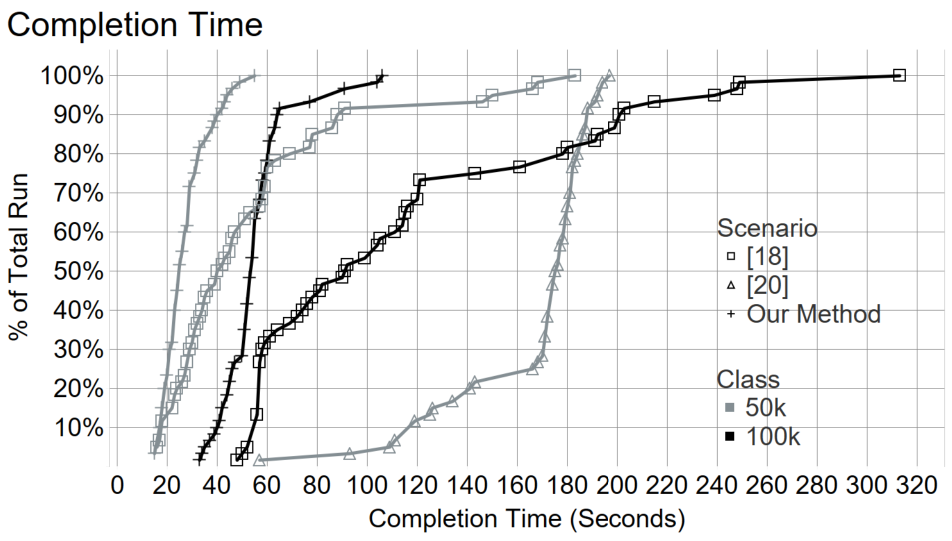

Figure 7. The compared completion time for the 5 scenarios

Table 7. The comparison table for completion time for the 5 scenarios

Completion time (seconds)				
Scenario	Average	Max	Min	Std. Dev (Pop)
1	27.3	55	15	8.86
2	55.3	106	33	14.23
3	53.1	183	16	38.82
4	112.5	313	48	63.46
5	166.2	197	57	28.11

completion time 27.3 seconds and 55.3 seconds for scenario 1 and 2 respectively. It is then followed by [18]'s architecture which has 53.1 and 112.5 seconds of completion time for scenarios 3 and 4 respectively. Finally, [20]'s has the longest average completion time of 166.2 seconds. The completion time for our method is faster than [18]'s architecture and [20]'s architecture as our architecture has faster response time. Thus, the time required to handle the requests is collectively faster. Hence, the completion time is faster.

As displayed in Figure 8, our method consumes the biggest CPU usage, followed by [18]'s architecture and finally [20]'s architecture. As seen in Table 8, our method consumes the biggest average CPU usage of 17.73% (for 50,000 requests) and 31.94% (for 100,000 requests). This is followed by [18]'s architecture, with CPU usage of 13.29% (for 50,000 requests) and 15.15% (for 100,000 requests). [20]'s consumes the least CPU usage of

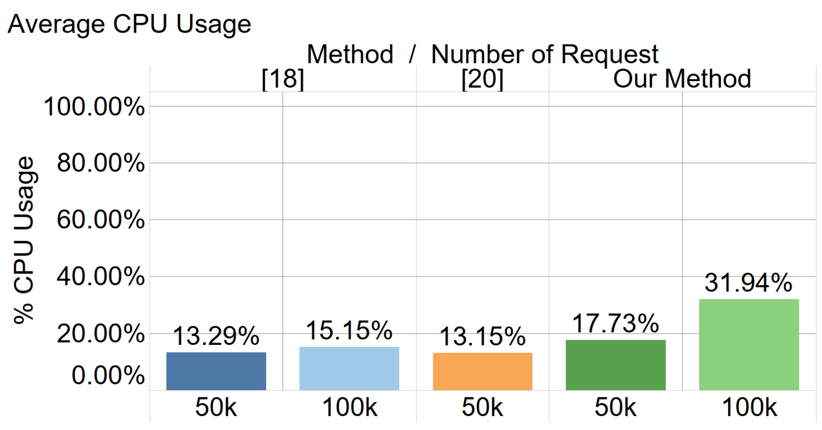

FIGURE 8. The compared CPU usage for the 5 scenarios

TABLE 8. The comparison table for CPU usage for the 5 scenarios

CPU usage (Percentage)				
Scenario	Average	Max	Min	Std. Dev (Pop)
1	17.73%	26.98%	7.24%	4.83%
2	31.94%	64.99%	9.06%	17.04%
3	13.29%	28.62%	11.75%	3%
4	15.15%	28.36%	8.35%	5.20%
5	13.15%	45.36%	12.14%	7.13%

13.15% (for 50,000 requests). The CPU usage is the lowest for [20]'s architecture as the cloud system has two virtual machines located in Singapore and two other virtual machines located in Oregon, with the weighted round-robin load balancer in Singapore, thus it is more resource effective compared to our proposed architecture. Our architecture has two master nodes in Oregon and Singapore, and two worker nodes in each region, thus it is heavier on the CPU usage compared to [18]'s architecture that only has 1 master node in Singapore. Our architecture consumes almost twice as much CPU time as the method of [18]'s architecture due to the added load balancing layer in our architecture. However, it is important to note that despite having the lowest CPU usage, [20]'s error rate is the highest. On the other hand, our method can achieve the lowest error rate despite consuming the highest CPU usage.

As displayed in the cumulative distribution function in Figure 9, our method has the lowest response time, followed by [18]'s architecture and finally [20]'s architecture. Our method also has the lowest average response times of 287 milliseconds (for 50,000 requests) and 639 milliseconds (for 100,000 requests), with a minimum response time of 2 milliseconds and maximum of 32810 for 50,000 requests, and a minimum of 2 milliseconds and maximum 68105 milliseconds for 100,000 requests. [18]'s architecture has an average response time of 1251 and 1218 milliseconds for 50,000 and 100,000 requests respectively. [20]'s architecture has the highest average response time of 9789 milliseconds, as seen in Table 9. Our proposed architecture has faster response time than [18] and [20]. [18]'s architecture does not regard which location the request is coming from. Hence, whether the request comes from Oregon or Singapore, the request will be handled in Singapore where the cloud system is. Therefore, taking the 50,000 requests, the response time is more significantly lower in our architecture, 287 milliseconds, than that in [18]'s architecture, 1251 milliseconds. In [20]'s architecture, every request will need to pass through

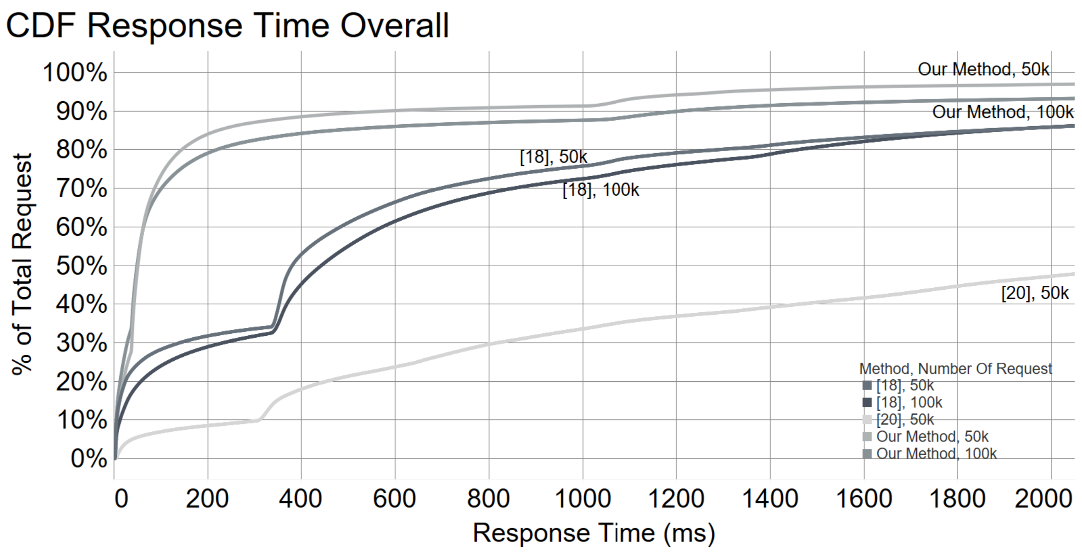

FIGURE 9. The comparison of response time for scenarios 1 to 5

TABLE 9. The comparison table for response time for scenarios 1 to 5

Response time (Milliseconds)				
Scenario	Average	Max	Min	Std. Dev (Pop)
1	287	32810	2	934
2	639	68105	2	2506
3	1251	148561	2	2808
4	1218	270112	2	2563
5	9789	133735	4	18757

the Singapore cloud system before it is sent to the designated system handling the request. Our architecture has the lowest average response time simply because it is message transferring measurements. When receiving a request, the data center must allocate the request efficiently to avoid congestion. Hence, the lower the response time, the better. Additionally, locality is a means for the application to be fault tolerant. When one service is down, the request can be passed on to another available service. This results in the application being fault tolerant.

In Figure 10, we limit the scope only to the scenarios with 50,000 concurrent requests, which are scenarios 1, 3 and 5. For each scenario, we split the data based on which region the requests come from, which are Singapore and Oregon.

In Figure 11, we also limit the scope only to the scenarios with 100,000 concurrent requests, which are scenarios 2 and 4. For each scenario, we also split the data based on which region the requests come from, which are Singapore and Oregon. Our method generally has responded to requests faster than [18]'s architecture. For the Oregon case, we can see the wider difference between [18]'s architecture and our architecture in the graph, where our architecture starts off from 0 seconds, meanwhile [18]'s architecture starts the lowest response time at around 350 milliseconds. This infers that our architecture can achieve lower response time on multi-regional cloud systems.

Based on our research, we found that our proposed method has the highest throughput, lowest completion time, lowest error rate and lowest response time. Our error rate is low because our data transmission is the fastest hence having lower chance of being intercepted

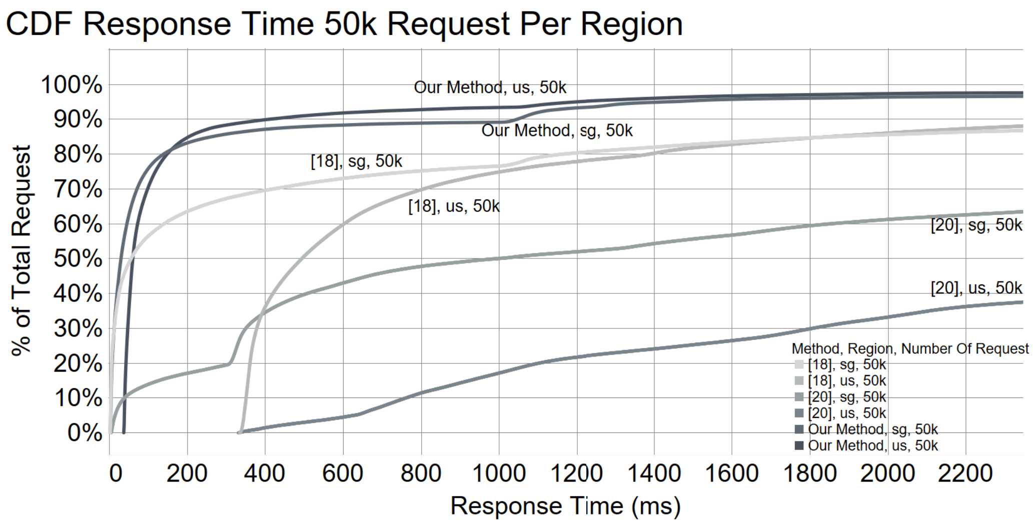

FIGURE 10. The response time comparison for 50,000 requests

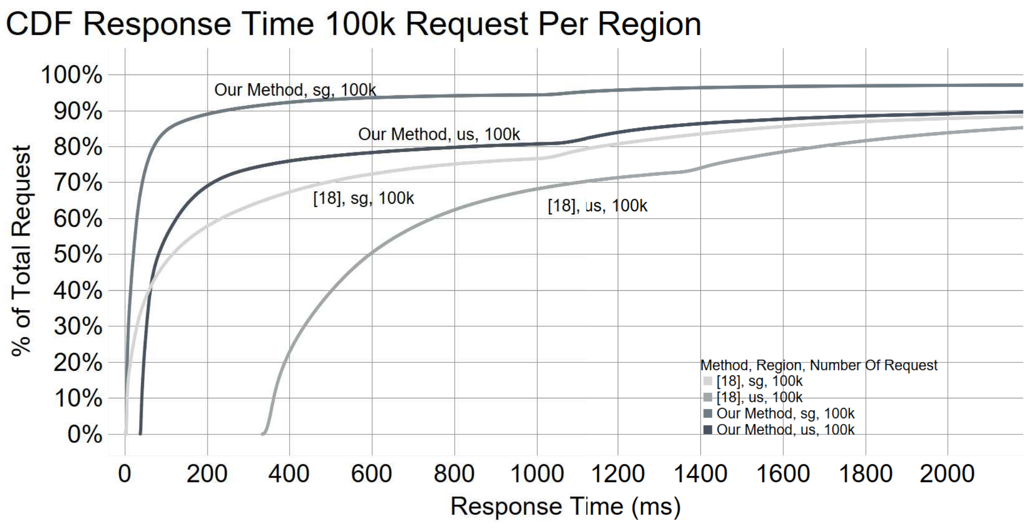

FIGURE 11. The response time comparison for 100,000 requests

before the data package is successfully delivered. Our proposed method has the highest CPU usage due to our process of needing to find the IP address in the Geo-IP database.

5. **Conclusions.**

This paper focuses on applying DNS load balancing on the previously reviewed architecture. We can conclude from our experiment that the application of DNS load balancing decreases the response time, decreases the completion time, and increases the throughput. This difference (compared to the two other architectures) is more visible at higher numbers of incoming requests or data. Also, the difference is more visible when the region is further away, as displayed in the Oregon case compared to the Singapore case.

In our future experimentation, we would like to enhance our proposed architecture with a feature for disaster recovery. It is to recover from an unexpected incidence.

Acknowledgment. This work is supported by the Directorate General of Higher Education, Ministry of Education, Culture, Research and Technology, Republic of Indonesia as a part of Penelitian Dasar Unggulan Perguruan Tinggi Research Grant to Binus University entitled "Sistem dan Aplikasi Portabel Pendeteksi Gangguan Irama Jantung Menggunakan Sinyal ECG Berbasis Machine Learning dan Cloud Computing" or "Portable ECG-Based Heart Arrhythmia Prediction System and Application Using Machine Learning and Cloud Computing" with contract number: 163/E4.1/AK.04.PT/2021.

REFERENCES

[1] N. A. Collop, W. M. Anderson, B. Boehlecke, D. Claman, R. Goldberg, D. J. Gottlieb, D. Hudgel, M. Sateia and R. Schwab, Clinical guidelines for the use of unattended portable monitors in the diagnosis of obstructive sleep apnea in adult patients. Portable monitoring task force of the American academy of sleep medicine, *Journal of Clinical Sleep Medicine*, vol.3, no.7, pp.737-747, 2007.

[2] N. A. Collop, S. L. Tracy, V. Kapur, R. Mehra, D. Kuhlmann, S. A. Fleishman and J. M. Ojile, Obstructive sleep apnea devices for out-of-center (OOC) testing: Technology evaluation, *Journal of Clinical Sleep Medicine*, vol.7, no.5, pp.531-548, 2011.

[3] J. M. Kelly, R. E. Strecker and M. T. Bianchi, Recent developments in home sleep-monitoring devices, *ISRN Neurology*, vol.2012, pp.1-10, 2012.

[4] M. De Zambotti, N. Cellini, A. Goldstone, I. M. Colrain and F. C. Baker, Wearable sleep technology in clinical and research settings, *Medicine Science in Sports Exercise*, vol.51, no.7, pp.1538-1557, 2019.

[5] M. Vanitha and P. Marikkannu, Effective resource utilization in cloud environment through a dynamic well-organized load balancing algorithm for virtual machines, *Computers Electrical Engineering*, vol.57, pp.199-208, 2017.

[6] R. Sharma and H. Reddy, Effect of load balancer on software-defined networking (SDN) based cloud, *2019 IEEE 16th India Council International Conference (INDICON)*, pp.1-4, 2019.

[7] F. F. Kherani and J. Vania, Load balancing in cloud computing, *International Journal of Engineering Development and Research*, vol.2, no.1, pp.907-912, 2014.

[8] S. Afzal and G. Kavitha, Load balancing in cloud computing – A hierarchical taxonomical classification, *Journal of Cloud Computing*, vol.8, no.1, p.22, 2019.

[9] S. Afzal and G. Kavitha, Optimization of task migration cost in infrastructure cloud computing using IMDLB algorithm, *2018 International Conference on Circuits and Systems in Digital Enterprise Technology (ICCSDET2018)*, 2018.

[10] K. Cao, Y. Liu, G. Meng and Q. Sun, An overview on edge computing research, *IEEE Access*, vol.8, pp.85714-85728, 2020.

[11] Y. Ai, M. Peng and K. Zhang, Edge computing technologies for Internet of Things: A primer, *Digital Communications and Networks*, vol.4, no.2, pp.77-86, 2018.

[12] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang and W. Zhao, A survey on Internet of Things: Architecture, enabling technologies, security and privacy, and applications, *IEEE Internet of Things Journal*, vol.4, no.5, pp.1125-1142, 2017.

[13] S. K. Mishra, B. Sahoo and P. P. Parida, Load balancing in cloud computing: A big picture, *Journal of King Saud University – Computer and Information Sciences*, vol.32, no.2, pp.149-158, 2020.

[14] S. K. Mishra, D. Puthal, B. Sahoo, S. K. Jena and M. S. Obaidat, An adaptive task allocation technique for green cloud computing, *The Journal of Supercomputing*, vol.74, no.1, pp.370-385, 2018.

[15] M. Arora, S. Das and R. Biswas, A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments, *Proc. of International Conference on Parallel Processing Workshop*, pp.499-505, 2002.

[16] I. R. Naredo and L. L. Pardavila, DNS load balancing in the CERN cloud, *Journal of Physics: Conference Series*, vol.898, 2017.

[17] N. Surantha, O. K. Utomo and S. M. Isa, High-performance and resource-efficient IoT-based sleep monitoring system, *2020 IEEE 91st Vehicular Technology Conference (VTC2020-Spring)*, pp.1-5, 2020.

[18] N. Surantha, V. V. Jansen, Wincent and S. M. Isa, Sleep quality monitoring system based on container orchestration, *ICIC Express Letters, Part B: Applications*, vol.11, no.11, pp.1011-1018, 2020.

[19] C. Qu, R. N. Calheiros and R. Buyya, Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing, *Concurrency and Computation: Practice and Experience*, vol.29, no.12, 2017.

[20] S.-L. Chen, Y.-Y. Chen and S.-H. Kuo, CLB: A novel load balancing architecture and algorithm for cloud services, *Computers Electrical Engineering*, vol.58, pp.154-160, 2017.
[21] J. Lim and D. Lee, A load balancing algorithm for mobile devices in edge cloud computing environments, *Electronics*, vol.9, p.686, 2020.
[22] R. K. Das and A. Lee, A study of load-balancing solutions of mobile cloud computing for next-generation mobile applications, *Proc. of the International Conference on Research in Adaptive and Convergent Systems*, Gwangju, Korea, pp.119-123, 2020.

Author Biography

Nico Surantha received his B.Eng. (2007) from Institut Teknologi Bandung, Indonesia. He received his Ph.D. degree from Kyushu Institute of Technology, Japan, in 2013. Currently, he serves as a assistant professor in Computer Science Department, Binus Graduate Program, Bina Nusantara University. His research interest includes wireless communication, health monitoring, network design, digital signal processing, system on chip design, and machine learning. He is an IEEE member.

Christopher Alvin is currently a full-time master track of information technology student at Bina Nusantara University, Indonesia. His research interest is in cloud computing and image recognition (computer vision).

Theresa Lusiana is currently a full-time master track of information technology student at Bina Nusantara University, Indonesia. Her research interest is in image recognition (computer vision) and computer network.