A Novel Weight-Assignment Load Balancing Algorithm for Cloud Applications

Adekunbi A. Adewojo[1][a] and Julian M. Bass[1][b]
[1] *University of Salford, The Crescent, Salford, Manchester, United Kingdom*
a.a.adewojo@edu.salford.ac.uk, j.bass@salford.ac.uk

Keywords: Cloud Computing, Load Balancing, Weight Assignment, Three-Tier Applications.

Abstract: Load balancing dynamically optimizes cloud resources and performance, and enhances the performance of applications deployed on cloud. We have chosen to investigate the class of cloud deployed web-based three-tier business applications. There is a problem with load balancing for this class of applications when they suffer from overload due to sudden flash crowds and resource failures. We propose a novel weight assignment load balancing algorithm to address this problem. Our approach utilises five carefully selected server metrics to efficiently distribute load among virtual machines. First, we validated our novel algorithm by comparing it with a baseline load-balancing algorithm and round-robin algorithm. Then, we experimentally evaluated our solution, by varying the number of user requests and carefully measuring response times and throughput. The experiments were performed on a private cloud environment testbed running OpenStack. Our experimental results show that our approach improves the response time of user requests by 5.66% compared to the baseline algorithm and 15.15% compared to round-robin algorithm in flash crowd scenario. In addition, while handling between 110% to 190% overload, our approach improved response times in all scenarios. Consequently, our novel algorithm outperforms the baseline and round-robin algorithms in overload conditions.

1 INTRODUCTION

Cloud computing, a platform for deploying web-based business applications, continues to gain rapid adoption in business and computing world. It is regarded as the recent answer to scalable and elastic computing. One of its appealing features is scalability (Qu et al., 2017; Grozev and Buyya, 2014b): an ability of the system's infrastructure to scale for handling growing workload requirements while maintaining a consistent performance.

A load balancer is commonly used to achieve consistent performance in applications deployed in cloud, while complementing the scalability of cloud. Load balancing in cloud computing is the process of efficiently and equally distributing workload across available resources. It plays a crucial role in achieving low latency, improving responsiveness, and maximising the utilisation (Hellemans et al., 2019) of resources in distributed systems. Load balancers use load balancing algorithm such as round-robin, least connections, hashing methods, and random algorithm to distribute workload requests among available resources. To ensure higher availability of services and responsiveness, specific applications such as three-tier web-based applications would benefit more from a targeted and improved load balancing algorithm and load balancer.

A web-based business application commonly experiences flash crowd: a rapid, fluctuating, and exponential request surge that happens because of increased users trying to access the application. The cloud's ability to scale, such that available resources can cater for this need, is highly essential. However, there is a problem with load balancing our class of applications when they experience flash crowds, and/or resource failures. The typical and common load balancing and auto-scaling strategies do not always suffice for these situations. Most times, these applications suffer performance degradation because of the inability of the load balancer to effectively distribute the workload or because the auto-scaling strategy was too slow to scale out resources or never responded at the required time.

Unfortunately, there has been limited research on load balancing algorithms for web-based three-tier business applications deployed on cloud. Even more

[a] https://orcid.org/0000-0003-1482-3158
[b] https://orcid.org/0000-0002-0570-7086

limited research on experimentally evaluating load balancing/distribution techniques using real cloud infrastructures. Most studies conducted in evaluating and developing load balancing techniques have focused on the use of simulation tools to evaluate these cloud models (Grozev and Buyya, 2014b; Shafiq et al., 2021; Kang et al., 2014; Tychalas and Karatza, 2020; Zomaya and Teh, 2001; Hellemans et al., 2019). This is in contrast to few evaluations done on real cloud infrastructure (Wang and Casale, 2014; Zhang et al., 2017; Chen et al., 2018).

In this paper, we propose a novel weight assignment load balancing algorithm. Our algorithm combines five carefully selected key server (Virtual Machine (VM)) metrics (thread count, network buffer, Central Processing Unit (CPU), Rapid Access Memory (RAM), and bandwidth utilisation) to properly distribute the workload of a web-based three-tier business application. It follows the monitor-analyse-plan-execute loop architecture commonly adopted by cloud-based systems (Zeng and Plale, 2014; Qu et al., 2017). This consistently distributes workload among available servers to maintain satisfactory response time and throughput. We validated our algorithm by comparing it to a baseline load balancing algorithm by (Grozev and Buyya, 2014b).

The key contributions of this research are:

- a novel hybrid-dynamic load distribution algorithm that improves response time, throughput, and scalability of web-based three-tier business applications; and
- an implementation of our algorithm evaluated in a private cloud test-bed

We organized the rest of this paper as follows: we briefly summarize related work and compare them to ours in Section 2. We motivate our research in Section 3. We illustrate the use case scenario of our solution in Section 4. We describe the architecture of our chosen application and some assumptions in Section 5. We introduce and detail our solution, its design and architecture in Section 6. We evaluate and present the results of our algorithm and benchmark algorithms in Section 9. Finally, we conclude the research in Section 10.

2 RELATED WORK

Some efforts have been made in developing and categorising load distribution techniques for the cloud. There are two categories of load balancing techniques, namely: static and dynamic techniques (Kumar and Kumar, 2019; Zomaya and Teh, 2001). Static techniques are typically used to distribute predictable load requests, while dynamic techniques are used to distribute unpredictable load requests.

Based on these categories, several authors have proposed different load balancing algorithms. Authors in (Shafiq et al., 2021) proposed a dynamic load balancing algorithm that addresses the VM isolation issue in the cloud through efficient task scheduling procedures. Their algorithm focused on optimising Quality of Service and resource allocation, thus improving the load balancing process in the cloud.

Authors in (Chen et al., 2017) also proposed a new paradigm for load balancing architecture and a new dynamic annexed load balancing technique that can be applied to both virtual web servers and physical servers. Their technique considers both server processing power and compute load to inform their load balancing algorithm. They concluded that their approach improved the mean response time of load balancing digital applications deployed on the cloud.

Apart from the above researches, authors in (Wang and Casale, 2014; Tychalas and Karatza, 2020; Devi and Uthariaraj, 2016; Sahu et al., 2013) all proposed dynamic load balancing algorithms that use a weighting mechanism to load balance cloud-based applications. They all considered specific server metrics such as CPU, RAM, and bandwidth in determining the dynamic weight of a particular physical server or VM. In contrast to these, our load balancing algorithm improves on those predefined server metrics by including thread count and network buffer to ensure that we can determine accurate and precise real time load of a server for effective routing of user's request.

Further researches were done to not only load balance workload but to improve on load balancing limitations such as single point of failure (Kumar and Kumar, 2019; Grozev and Buyya, 2014b; Cruz et al., 2019), scalability, limitation in sensing uncertainties (Zhang et al., 2017), excessive overhead and re-routing (Zhang et al., 2017).

Building upon research works that are focused on load balancing limitations, authors in (Zhang et al., 2017) proposed a resilient data centre load balancer called Hermes, which is resilient to traffic dynamics, topology asymmetry, and failures. It leverages comprehensive sensing to detect path conditions and apply correct re-routing of requests to the server.

Authors in (Cruz et al., 2019) proposed a task mapping algorithm to improve communication and load balancing in clusters of multicore systems. They used the eagermap algorithm to determine task mappings, which is based on a greedy heuristic to match application communication patterns to hardware hierarchies and also consider task load. They claim that their algorithm design alleviates the single point of failure problem. Our load balancing algorithm further improves on the above-mentioned limitations-amelioration technique by creating an architecture that avoids single point of failure, reduces excessive re-routing by using HAProxy, and sensing uncertainties by using selected server metrics that determine a server's capacity.

Authors in (Grozev and Buyya, 2014b) proposed an adaptive and dynamic resource provisioning and load distribution algorithms for multi-cloud to heuristically optimize overall cost and response delays without violating essential legislative and regulatory requirements.

Our solution is different from the above approaches described above. Although it is similar to the approaches of Grozev (Grozev and Buyya, 2014b) and Wang (Wang and Casale, 2014), but it combines carefully selected key server metrics that are specific for accommodating our chosen class of application. We also adopt an architecture composed of multiple available load balancers to alleviate the problems associated with load balancing in the cloud. Furthermore, similar to the research work of Grozev (Grozev and Buyya, 2014b), our proposed load balancing algorithm can be extended to work in a multi-cloud setting.

In addition, evaluation of the above researches and those of (Hellemans et al., 2019; Zomaya and Teh, 2001; Elgedawy, 2015) were all done using simulation tools and environment. We note that there is little research (Chen et al., 2018) on evaluating cloud computing models by completely using real cloud infrastructures. Most times, a simplified real infrastructure experiment is done to complement simulated experiments as seen in (Wang and Casale, 2014; Grozev and Buyya, 2014b; Zhang et al., 2017).

However, we argue that simulation experiments rely heavily on parameters to be accurate, so there is a challenge of knowing what an accurate parameter value should be. The consequence of this is an incorrect simulation result when the parameters are not right. Meanwhile, running experiments on real cloud infrastructure do not require parameterisation, we get the real behaviour of all resources used and this produces a more realistic result. Our research validates our baseline algorithm, compares our algorithm to the baseline and round-robin algorithms, and tests our novel algorithm using a real cloud test environment.

3 MOTIVATION

Research (Grozev and Buyya, 2014b; Chen et al., 2017; Tychalas and Karatza, 2020) shows that standard load balancing technique does not suffice for most applications deployed on the cloud. How much more, web applications that commonly experiences flash crowds and sometimes resource failures. These studies corroborate the fact that there is need for a dynamic and real-time capacity focused load balancing algorithm for our class of application.

An approach to alleviate issues associated with a standard load balancing algorithm is to incorporate into the algorithm the key factors that affect the real-time load of a VM. Our research identified the following key server metrics namely: CPU utilisation, memory utilisation, network bandwidth, number of threads running, and network buffers, as the most relevant determinants of a VM's real-time capacity and load. Incorporating these factors will help a load balancer to make decisions based on the current capacity and real-time load of a VM, and therefore better utilise available resources and provide improved performance.

Our proposed algorithm is an enhancement of a load balancing algorithm by (Grozev and Buyya, 2014b). It incorporates the dynamic load balancing technique, and the above-mentioned server factors to calculate the weight of a VM. It assigns each VM a weight based on the current utilisation, so the weight of each VM and the probability of each VM's usage dynamically changes during runtime after evaluating its current state. We implement this idea in our novel Load Balancing Algorithm as depicted in 1.

4 USE CASE SCENARIOS

Our load balancing algorithm will be effective in the following scenarios:

4.1 Flash Crowds

Flash crowds are traffic spikes that commonly affects web applications (Qu et al., 2017; Ari et al., 2003; Le et al., 2007). They happen without any prior notice and may become difficult to manage. Commercial cloud providers use auto-scaling services to combat flash crowds. These services launch new VMs after the application has experienced high load for a specific period of time set by the user. Our algorithm can easily complement the role of an auto-scaler in any cloud based deployment, by effectively balancing application workload across available servers, thereby maintaining consistent performance that do not fall below predefined service level agreement (SLA) before resources are provisioned.

4.2 Resource Failures

Cloud resource failure occurs when any of the components of a cloud environment fails to function as it is intended to. According to (Priyadarsini and Arockiam, 2013; Prathiba and Sowvarnica, 2017), the three most common resource failures in any cloud environment are hardware, virtual machines and application failures. Any of this cloud resource failure can happen suddenly, and immediately result into degradation of service performance or total loss of service. Most cloud providers use the intervention of an auto-scaler and load balancer to mitigate the effect of any such failures. However, the time required for an auto-scaler to provision new resources will usually result into performance degradation if existing and new requests are not well managed, so our proposed load balancing technique will definitely help to maintain acceptable service performance while the auto-scaler is provisioning resource.

5 APPLICATION ARCHITECTURE AND REQUIREMENTS

Our target applications are three-tier web-based business applications deployed on cloud. The three-tier architecture is the most popular implementation of a multi-tier architecture (Amazon, 2021). A three-tier web application is an interactive application that can be accessed over the internet. Its architectural layers are divided into three, namely: presentation, domain, and data layer.

Our approach is the foundation to tenanted multi-cloud load distribution, hence it will require requests that can be processed by application replicas deployed in more than one data center. This involves some important factors, and the most important factor for this research is session continuity.

Session continuity ensures that end user sessions, established over any access networks, will not lose connection or any internal state even when different servers process the user requests. Stateless applications, such as search engines, does not save client data generated in one session for use in the next session with that client. Also, stateless applications can easily scale because they can be deployed across multiple servers without issues while ensuring session continuity. All these properties of stateless applications implicitly satisfy the requirement of session continuity. On the other hand, stateful applications require persistence of end-user IP and session to a specific server. Therefore, stateful applications and services cannot be managed by our approach.

6 THE PROPOSED APPROACH

6.1 Load Balancing Architecture

Our approach to designing a hybrid-dynamic load balancing algorithm using a novel weight-assignment policy is depicted in Figure 1. We co-locate the load balancer and our load balancing service in the same data centre to achieve fast detection of flash crowds and perform quick adaptations. The load balancing service comprises a monitoring module, that constantly monitors incoming requests and the status of available resources to detect application or server overload; and a control module to modify the weight of each VM to accommodate request workload.

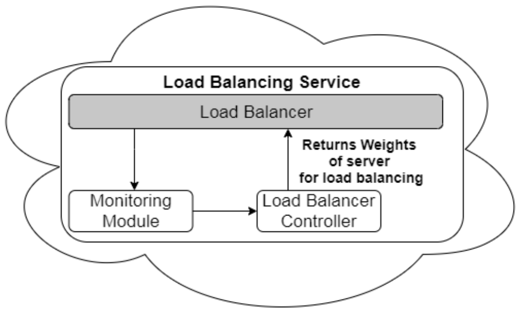


Figure 1: Proposed Load Balancing Architecture

6.2 Overall Architecture

We depict the proposed overall architecture in Figure 2. We extend the conventional three-tier architectural pattern with one additional layer of components. The proposed load balancing layer comprises three distinct components: HAProxy load balancer, our load balancing service, and an auto scaler.

Users are presented with an entry point that consists of VMs that host load balancing service that distribute requests to application servers. The domain layer is replicated over multiple Application Servers and are hosted in separate VMs in our experimental cloud test-bed. This enables the domain layer to be scaled horizontally. The data layer also consists of

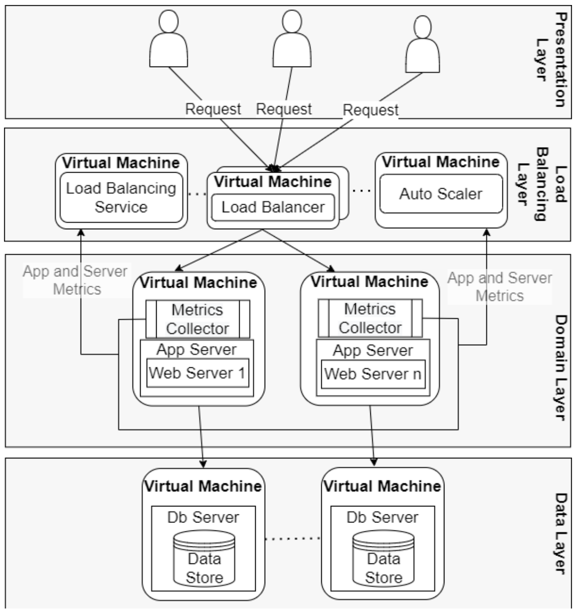


Figure 2: Proposed Overall Layered Architecture in a Single Data Center

multiple database servers installed across VMs.

6.3 Design and Deployment Architecture

A good load balancing algorithm should reduce response time, increase throughput and improve utilisation of resources while enhancing system performance. Furthermore, a suitable load balancing technique should consider different metrics to make it relevant for applications whose size and needs may increase suddenly, leading to the use of more resources (Kumar and Kumar, 2019).

The issues and limitations discussed in section 2 highlights areas that need improvement in a load balancing algorithm. We discuss relevant techniques used in our architecture that mitigate these issues as follows:

- Scalability: Ability of a load balancer (or algorithm) to perform load balancing with any finite number of nodes (Kansal and Chana, 2012). Our proposed architecture is scalable because our load balancing algorithm is built on a proven load balancing system – HAProxy 2.4.2-1 (HAProxy, 2021). We have also tested our solution using a varied number of heterogeneous VMs ranging from one to eight VMs to test scalability. Our solution has proven to load balance workload with finite number of nodes.
- Fault Tolerance: Ability of a load balancer to continue to deliver services despite failure (Shah et al., 2017) of a load balancer or a server. We mitigate the issue of single point of failure by creating multiple load balancing front-ends and a standby application server, as shown in Figure 2. Our approach uses the high availability keep alive technology of HAProxy to monitor the services and automatically switch over to a standby server. To make the load balancer fault-tolerant, we create a floating IP that can be moved between load balancers; if the primary load balancer goes down, the floating IP moves to the second load balancer automatically, allowing service to continue.
- Reduced Overhead and Latency: Overhead refers to the extra time taken by a load balancer when distributing workload, which increases communication cost (Zomaya and Teh, 2001; Shah et al., 2017). The extra overhead generated by our algorithm is negligible because it uses a decentralised approach to collect and maintain the server's metrics, as recommended by (Zomaya and Teh, 2001). Our architecture includes VMs with glances (Hennion, 2021) agents installed on each of them. It uses the RESTful Application Programming Interface (API) of glances to capture server metrics that serves as inputs to our load balancing service. To calibrate performance indicator, these agents also send server metrics to InfluxDB — a time series database.
- Server Metrics: Ability of a load balancing algorithm to make use of relevant server metrics in deciding how to balance the workload of an application (Chen et al., 2017). Our algorithm combines carefully selected key server metrics that determine a server load as evidenced in these studies (Grozev and Buyya, 2014b; Tychalas and Karatza, 2020; Wang and Casale, 2014; Sahu et al., 2013). We argue that aside from the common metrics of CPU, and Memory utilisation, network bandwidth utilisation and Thread Count are vital server metrics because they determine how efficient data is transmitted in and out of a system; and summarises the number of concurrent requests happening in the server at a particular time respectively. These metrics help us to understand what the general load of a server looks like from a request level, and the amount of load on the server when running multiple threads.

7 PROPOSED LOAD BALANCING SERVICE

This section describes our novel weight assignment technique and load balancing algorithm.

7.1 Proposed Weighting Technique

The weighting technique for our novel load balancing algorithm improves the research of (Chen et al., 2018) and (Chen et al., 2017). We made use of four of our proposed metrics in the weighting calculation, while we used the fifth metric in the proposed algorithm.

To compute the weight of a VM, $W(X_k)$, we first calculated its real-time load, $Lr(X_k)$, in equation (1), using the following server metrics. M_k represents Memory utilisation, B_k represents Bandwidth utilisation, C_k represents CPU utilisation, T_k represents Thread Count and NB_k represents Transmission Control Protocol (TCP) network buffer/queues. These parameters are derived as percentages and must be converted to integer values by dividing it by 100.

$$\begin{aligned} Lr(X_k) = (e_1 * (C_k/100)) + (e_2 * (M_k/100)) + \\ (e_3 * (B_k/100)) + (e_4 * (T_k/100)) \end{aligned} \tag{1}$$

To acknowledge the different influences of each metric factor on a VM, we introduce a weight factor to the metrics used in equation (1). e_1, e_2, e_3, e_4 represents weight of CPU, RAM, bandwidth, and thread count respectively. The sum of them is 1.

We carefully chose experimental proven values for each of the weight factors. We fitted these data in experiments, and they represent best fit values of the different influence of our chosen server metrics of an application server for a three-tier web-based business application. The values chosen are $w = 0.5, 0.3, 0.15, 0.05$ for CPU, RAM, bandwidth, and thread count respectively. CPU utilisation has the highest weight factor because our class of application is processor intensive. Memory utilisation has the next highest weight factor because it makes the VM become quickly unresponsive when being heavily used, even when other metrics are not fully utilised.

We did not include network buffer in equation (1), and as a result; we did not assign a weight factor to it. This is because this factor is used as a check of network utilisation/availability or how busy the network is at any particular time, and its influence on a server is constant. However, we use network buffer in our load balancing algorithm to continuously monitor available network availability when calculating real-time load.

We define the average value of all available application server VM load as a threshold for load comparison; this is represented in equation (2), where n represents the number of all available application server VMs.

$$Lr_{th} = \frac{\sum Lr(X_k)}{n} \tag{2}$$

We further improved and analysed our weight calculation as follows: if $Lr(X_k) \leq Lr_{th}$, this shows that the application server VM's load is relatively small, so we should increase its weight. If it is the opposite, we decrease the weight. Therefore, we define a modification parameter δ as shown in equation (3) using the weighted averages.

$$\delta = \frac{Lr(X_k)}{\sum Lr(X_k)} \tag{3}$$

The least utilised application server VM will return the lowest real time load, but our load balancer, HAProxy, functions by appropriating a bigger weight value to the least utilised VM in a set of VMs. So to achieve this, we computed the inverse of the real-time load, $Lr(X_k)$, in equation (4). Doing this will make the least utilised VM's real load time to have the biggest value.

Finally, because the weight used by HAProxy is bounded for real integer values, the weight must be a whole number; otherwise it will not be consistent with the original intention. Our algorithm implementation rounds up any decimal value that is greater than eight to make the value an integer. We represent the complete weight equation in (4).

$$W(X_k) = \begin{cases} (\frac{1}{Lr(X_k)} + \delta), Lr(X_k) \leq Lr_{th} \\ (\frac{1}{Lr(X_k)} - \delta), Lr(X_k) \geq Lr_{th} \end{cases} \tag{4}$$

7.2 Proposed Load Balancing Algorithm

Our proposed novel algorithm, Algorithm 1 abstracts the overall flow of the weighting and load balancing algorithm. It defines a hybrid—dynamic load balancing algorithm; it computes the utilisation of each server every two seconds and then assigns a weight to each server during run-time using our proposed weighting technique. The input parameters of the algorithm are:

- Thc—CPU threshold;
- Thr—RAM threshold;
- $Thbw$—Bandwidth threshold;
- Th_tr—Thread count threshold;

- VM_{as}—list of currently deployed application server VMs;

As the first step in the algorithm, we receive and set the overall threshold for the input parameters. We experimentally set values of $80\%, 80\%, 80\%, 85\%$ for CPU, RAM, Bandwidth, and Thread count respectively, by performing a profiling tests on one medium application server VM using synthetic workload generated according to real requests. We used a predefined SLA of 90% requests should be replied within one second as recommended by (Qu et al., 2017) to determine the average utilisation percentage.

To simplify our algorithm, we extracted the logic for checking TCP network buffer NB_k in a separate function called **TCPBufferOverloaded()**. This function uses the netstat command to check if there is a TCP socket for which any of the ratios of the buffer sizes, Recv-Q and Send-Q values are greater than 0.9. A ratio greater or equal to 1 mean the network buffer is overloaded and requests will not arrive promptly at that VM.

For each of the VM, we retrieved the utilisation values of our chosen VM metrics. In line 2, our algorithm loops through a list of available application server VMs and compares each utilisation metric against the set threshold for the current VM. After that, it computes the weight as defined in equation (4) and checks the TCP network buffer NB_k of each application server VM. It assigns a weight to each VM whose TCP network buffer is not overloaded as shown in line 9; otherwise, if the VM is fully utilised (100%), it assigns a weight of 0 to the VM as shown in line 11 so that the VM does not accept any more incoming request until the utilisation is lower or equal to the set threshold.

Finally, requests were distributed to each server in turns in line 14, according to their weights. This distributes server load proportionally based on a VM's real-time capacity.

8 EXPERIMENTAL TEST-BED

We illustrate our experimental test-bed in Figure 3. It consists of twelve heterogeneous VMs as described in Table 1.

We deployed Apache Jmeter; a workload simulator, on an external machine. We deployed HAProxy server along with our load balancing service on two VM instances. We launched five medium VM instances of application servers and four large instances of database server.

Algorithm 1: Novel Load Balancing Algorithm.

```
   Input: $s_i$, $Thc$, $Thr$, $Thbw$, $Th_tr$, $VM_{as}$
1  RetrieveAllocateToInputThresholdValues ();
2  for each VM, $vm_i \in VM_{as}$ do
3      $Utl_{cpu} \leftarrow$ CPU utilisation of $vm_i$;
4      $Utl_{ram} \leftarrow$ RAM utilisation of $vm_i$;
5      $Utl_{bw} \leftarrow$ Bandwidth utilisation of $vm_i$;
6      $UtlThreadCount \leftarrow$ Threadcount of $vm_i$;
7      if ($Utl_{cpu}$ ¡ $Thc$ & $Utl_{ram}$ ¡ $Thr$ & $Utl_{bw}$ ¡ $Thbw$ & $Utl_{ThreadCount}$ ¡ $Th_tr$ & !TCPBufferOverloaded ()) then
8          $W(X_k) \leftarrow$ CalculateWeightbyutilisation ($Utl_{cpu}$, $Utl_{ram}$, $Utl_{bw}$, $UtlThreadCount$, $VM_{as}$, $vm_i$);
9          assignweighttoVM ($vm_i, W(X_k)$);
10     else
11         assignweighttoVM ($vm_i, 0$);
12     end
13 end
14 HAProxyAssignRequest ($s_i, VM$)
```

Table 1: VM Capacity.

	m1.medium	m1.large
VCPUs	4	8
RAM	4GB	8GB
Disk Size	40GB	80GB

8.1 Case Study Application

Our case study application is a stateless, three-tier, open source multi-tenant E-commerce application used to buy and sell various products on the Internet. It is built using Orchard Core framework coupled with Elastic search for implementing its search engine. The application is supported by MySQL database. Because this research does not consider data deployment, we focused on the application tier.

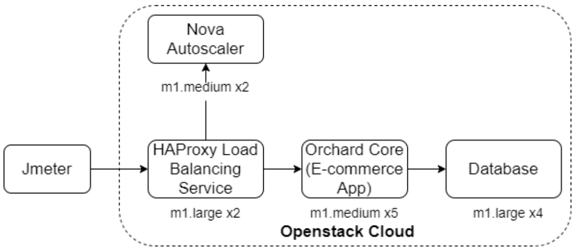


Figure 3: Experimental Test-Bed

8.2 Workload

To simulate realistic user requests and location, we hosted JMeter on a non-cloud machine. First, we profile our application instance to determine the amount of requests that can be successfully handled within one second. Second, we profile our application instance again to determine the amount of requests that can be handled, if our SLA stipulates that 90% of requests must be handled within one second, this SLA mechanism was recommended by (Qu et al., 2017).

An average combination of the two profile showed that our application server instance can handle 90 requests per second. Based on this profiling, we created our workload to emulate a Poisson distribution because of the nature of user request arrival, as indicated by (Chlebus and Brazier, 2007). We used Jmeter to send requests to our deployed application using our workload model. We repeated each experiments three times and took the average of the repeated tests in all our evaluations.

8.3 Baseline Application and Evaluation

To test the performance of our solution, we compared our approach with our baseline algorithm by (Grozev and Buyya, 2014b) and a standard load balancing (round-robin) algorithm. Our evaluation sought to characterise prominent load balancing performance parameters, namely response times and throughput (Zomaya and Teh, 2001; Shah et al., 2017).

9 PERFORMANCE ANALYSIS AND RESULTS

In this section, we analyse and compare the performance offered by our algorithm and the benchmarks.

9.1 Performance under Resource Failures

We first validated our algorithm to determine if it will effectively distribute requests by first simulating over utilized VMs as depicted in Figure 4. Our algorithm assigned correctly a weight of 0 to overutilised VMs and then distributed load across available VM based on their assigned weight.

To test fault tolerance and scalability, we performed tests in resource failure situations. In these experiments, we remove some VMs from the available VMs at 300ms time point. This process creates hardware failure. We added the VMs back to the available

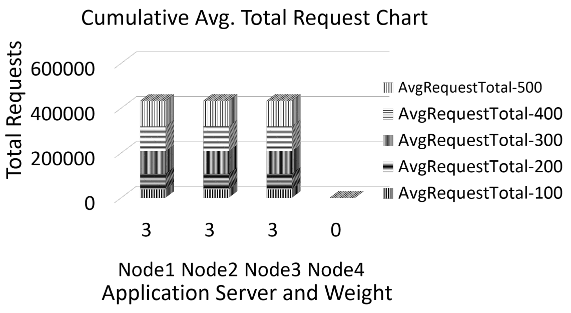


Figure 4: overutilised VM

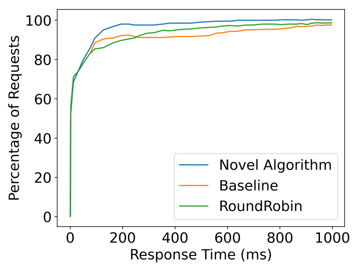
VM pool after five seconds to emulate recovery from failure.

We compared the performance of our algorithm with the benchmark algorithms. The baseline algorithm by (Grozev and Buyya, 2014b) suffered performance degradation, with very high response times, such that the percentage of requests it could handle declined. By the time the number of hardware failures increased to three, the algorithm's performance could only attend to 85% of requests within one second, which is less than the SLA of 90% of requests to be completed within one second. The round-robin algorithm performed worst, because the amount of requests handled were much less than the SLA in all three scenarios of server failures. Figures 5a, 5b, and 5c depicts the performances of each of the algorithms during the three different server failures. Our algorithm did not violate the SLA, and it maintained low failed requests as Figure 6. These result proved that our algorithm can distribute requests effectively even during server failures.

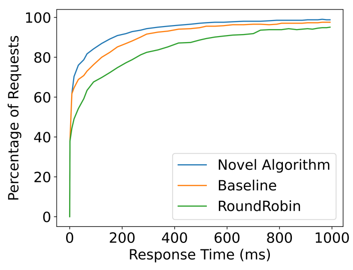
9.2 Performance under Flash Crowds

We tested the performance of our algorithm when the application experiences flash crowds. First, we saturated three VMs to make them 100% utilised for a period of five seconds. Our algorithm consistently distributed the requests, maintained SLA and the average response time was kept low and became lower by the time the auto-scaler provisioned a new VM. The baseline algorithm by (Grozev and Buyya, 2014b) could not evenly distribute requests and had more failed requests within four seconds of the flash crowd. The round-robin algorithm became unresponsive because it had a queue of requests trying to access the application servers.

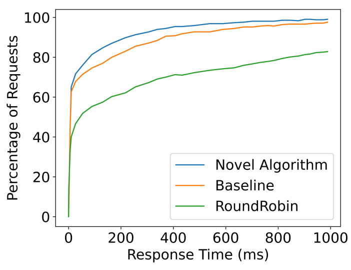
Secondly, we emulated flash crowds by updating our workload to exponentially increase requests for a period of 300ms every five seconds within 1 minute. The peaks of the flash crowd range from 110% to 190% of the normal workload. Our algorithm still



(a) 1 Server Failure.



(b) 2 Server Failures.



(c) 3 Server Failures.

Figure 5: Cumulative Distribution Values of One Server Failure.

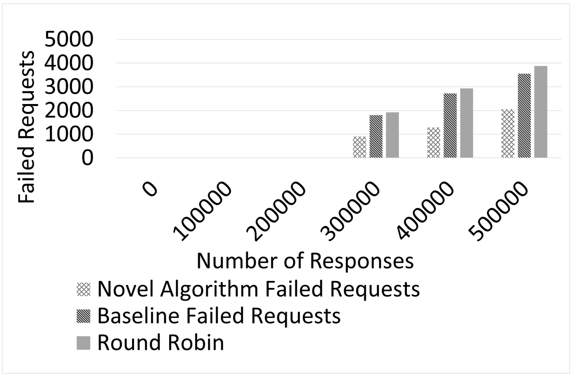


Figure 6: Failed Request Chart — Error Rates

performed better than the other two algorithms as displayed in Figures 7a, 7b, and 7c. Requests were distributed without violating SLA unlike the benchmark algorithms, they violated SLA by the time flash crowds exceed 130% of the normal load. This confirms our algorithm can accommodate changing user requests while maintaining the defined SLA. This further validates our choice of carefully selected server metrics, because the awareness of these metrics improves the accuracy of determining the capacity of VMs handling these requests.

10 CONCLUSIONS AND FUTURE WORK

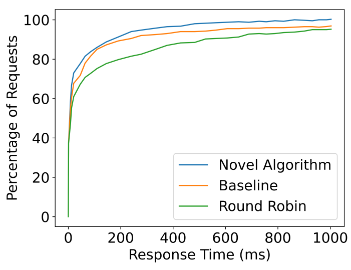
In this paper, we proposed a novel weight assignment load balancing algorithm for web-based three-tier applications deployed on the cloud. This class of application commonly suffer from overload due to sudden flash crowds and resource failures. To address this problem, our approach utilised five carefully selected server metrics namely: thread count, CPU, RAM, and network bandwidth utilisation. This approach effectively determines the real-time capacity of a VM, thereby allowing an efficient distribution of load. Our novel algorithm employed the server metrics to determine the weight of a VM, and the algorithm forms part of a load balancing service (HAProxy and autoscaler) to properly distribute load among virtual machines. In addition, we adopted a highly available deployment architecture to overcome challenges common with load balancing architectures such as single point of failure, reliability, and scalability. Furthermore, we deployed our load balancing service on the same data centre to enable quick adaptation to changes in the system.

We implemented and evaluated our novel algorithm on a private cloud data centre running OpenStack. First, we validated our novel algorithm by comparing it to a baseline load balancing algorithm (Grozev and Buyya, 2014a) and a standard algorithm (round-robin). Second, we sent varied user requests using a workload pattern in Jmeter, to deployed applications and carefully measured widely accepted performance parameters – response time and throughput. The obtained results showed that our approach is able to distribute the workload of our chosen class of application while maintaining a widely accepted SLA of 90% of requests to be completed within one second. The response times were improved by 5.66% and 15.15% compared to the baseline algorithm and round-robin algorithm in flash crowd scenarios.

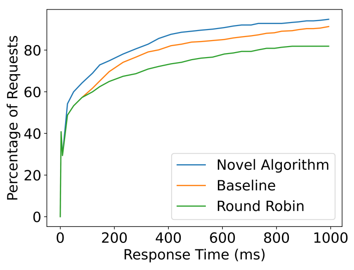
As future work, we will improve on some limiting factors of this research. We aim to implement our novel algorithm on one or more public cloud, so that we can compare and validate the performance of the algorithm in different clouds. Furthermore, we will extend our load balancing algorithm to improve resource allocation in multi-cloud deployment of web applications.

ACKNOWLEDGEMENTS

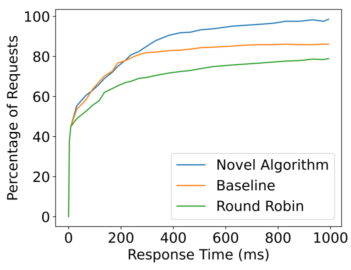
We thank Daniel MacDonald, for helping us to install, and commission our private cloud environment.



(a) 135 req/s flash crowd.



(b) 188 req/s flash crowd.



(c) 240 req/s flash crowd.

Figure 7: Cumulative Distribution of Flash Crowd.

REFERENCES

Amazon (2021). Aws serverless multi-tier architectures with amazon api gateway and aws lambda aws whitepaper.

Ari, I., Hong, B., Miller, E. L., Brandt, S. A., and Long, D. D. (2003). Managing flash crowds on the internet. In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003.*, pages 246–249. IEEE.

Chen, S.-L., Chen, Y.-Y., and Kuo, S.-H. (2017). Clb: A novel load balancing architecture and algorithm for cloud services. *Computers & Electrical Engineering*, 58:154–160.

Chen, Z., Zhang, H., Yan, J., and Zhang, Y. (2018). Implementation and research of load balancing service on cloud computing platform in ipv6 network environment. In *Proceedings of the 2nd International Conference on Telecommunications and Communication Engineering*, pages 220–224.

Chlebus, E. and Brazier, J. (2007). Nonstationary poisson modeling of web browsing session arrivals. *Information Processing Letters*, 102(5):187–190.

Cruz, E. H., Diener, M., Pilla, L. L., and Navaux, P. O. (2019). Eagermap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems. *ACM Transactions on Parallel Computing (TOPC)*, 5(4):1–24.

Devi, D. C. and Uthariaraj, V. R. (2016). Load balancing in cloud computing environment using improved weighted round robin algorithm for nonpreemptive dependent tasks. *The scientific world journal*, 2016.

Elgedawy, I. (2015). Sultan: A composite data consistency approach for saas multi-cloud deployment. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 122–131.

Grozev, N. and Buyya, R. (2014a). Inter-cloud architectures and application brokering: taxonomy and survey. *Software: Practice and Experience*, 44(3):369–390.

Grozev, N. and Buyya, R. (2014b). Multi-cloud provisioning and load distribution for three-tier applications. *ACM Trans. Auton. Adapt. Syst.*, 9(3):13:1–13:21.

HAProxy (2021). Haproxy technologies — the world fastest and most widely use load balancing solution.

Hellemans, T., Bodas, T., and Van Houdt, B. (2019). Performance analysis of workload dependent load balancing policies. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(2):1–35.

Hennion, N. (2021). Glances an eye on your system. a top/htop alternative for gnu/linux, bsd, mac os and windows operating systems.

Kang, S., Veeravalli, B., and Mi Aung, K. M. (2014). Scheduling multiple divisible loads in a multi-cloud system. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 371–378.

Kansal, N. J. and Chana, I. (2012). Cloud load balancing techniques: A step towards green computing. *IJCSI International Journal of Computer Science Issues*, 9(1):238–246.

Kumar, P. and Kumar, R. (2019). Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Computing Surveys (CSUR)*, 51(6):1–35.

Le, Q., Zhanikeev, M., and Tanaka, Y. (2007). Methods of distinguishing flash crowds from spoofed dos attacks. In *2007 Next Generation Internet Networks*, pages 167–173. IEEE.

Prathiba, S. and Sowvarnica, S. (2017). Survey of failures and fault tolerance in cloud. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)*, pages 169–172. IEEE.

Priyadarsini, R. J. and Arockiam, L. (2013). Failure management in cloud: An overview. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(10):2278–1021.

Qu, C., Calheiros, R. N., and Buyya, R. (2017). Mitigating impact of short-term overload on multi-cloud web applications through geographical load balancing. *concurrency and computation: practice and experience*, 29(12):e4126.

Sahu, Y., Pateriya, R., and Gupta, R. K. (2013). Cloud server optimization with load balancing and green computing techniques using dynamic compare and balance algorithm. In *2013 5th International Conference and Computational Intelligence and Communication Networks*, pages 527–531.

Shafiq, D. A., Jhanjhi, N. Z., Abdullah, A., and Alzain, M. A. (2021). A load balancing algorithm for the data centres to optimize cloud computing applications. *IEEE Access*, 9:41731–41744.

Shah, J. M., Kotecha, K., Pandya, S., Choksi, D., and Joshi, N. (2017). Load balancing in cloud computing: Methodological survey on different types of algorithm. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 100–107. IEEE.

Tychalas, D. and Karatza, H. (2020). An advanced weighted round robin scheduling algorithm. In *24th Pan-Hellenic Conference on Informatics*, pages 188–191.

Wang, W. and Casale, G. (2014). Evaluating weighted round robin load balancing for cloud web services. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 393–400. IEEE.

Zeng, J. and Plale, B. (2014). Multi-tenant fair share in nosql data stores. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 176–184. IEEE.

Zhang, H., Zhang, J., Bai, W., Chen, K., and Chowdhury, M. (2017). Resilient datacenter load balancing in the wild. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 253–266.

Zomaya, A. Y. and Teh, Y.-H. (2001). Observations on using genetic algorithms for dynamic load-balancing. *IEEE transactions on parallel and distributed systems*, 12(9):899–911.