Multi-cloud Load Distribution for Three-tier Applications

Adekunbi A. Adewojo[a] and Julian M. Bass[b]
University of Salford, The Crescent, Salford, Manchester, U.K.

Keywords: Cloud Computing, Multi-cloud, Load Balancing, Algorithm, Three-tier Applications.

Abstract: Web-based business applications commonly experience user request spikes called flash crowds. Flash crowds in web applications might result in resource failure and/or performance degradation. To alleviate these challenges, this class of applications would benefit from a targeted load balancer and deployment architecture of a multi-cloud environment. We propose a decentralised system that effectively distributes the workload of three-tier web-based business applications using geographical dynamic load balancing to minimise performance degradation and improve response time. Our approach improves a dynamic load distribution algorithm that utilises five carefully selected server metrics to determine the capacity of a server before distributing requests. Our first experiments compared our algorithm with multi-cloud benchmarks. Secondly, we experimentally evaluated our solution on a multi-cloud test-bed that comprises one private cloud, and two public clouds. Our experimental evaluation imitated flash crowds by sending varying requests using a standard exponential benchmark. It simulated resource failure by shutting down virtual machines in some of our chosen data centres. Then, we carefully measured response times of these various scenarios. Our experimental results showed that our solution improved application performance by 6.7% during resource failure periods, 4.08% and 20.05% during flash crowd situations when compared to Admission Control and Request Queuing benchmarks.

1 INTRODUCTION

One of the attractive features of the cloud is its ability to dynamically expand or shrink the amount of resources using auto-scaling services. Despite the ability of cloud to rapidly detect workload changes and auto-scale, it requires a considerable amount of time. Experimental research on Virtual Machine (VM) startup shows that it takes between 50 and 900 seconds to boot up a VM depending on the size, model, cost and operating systems (Qu et al., 2017). This delay in start-up often result into performance degradation and may even result in temporary system unavailability if it is not well managed.

Web applications commonly suffer from rapid surges in user requests. The terminology for this common scenario is flash crowds (Qu et al., 2017); and it can occur with little or no warning. This sudden burst of legitimate network activity are usually responsive to traffic control, and are web traffic type. This is unlike distributed denial of service attack(DDOS) which is usually unresponsive to traffic control, and occurs as any traffic type (Wang et al., 2011). In addition, sudden resource failure can lead to overload or complete downtime of cloud deployed web applications. Cloud providers usually mitigate flash crowds cases by using an auto-scaler to dynamically provision enough resources. However, because these situations occur rapidly, the auto-scaler cannot timely provision enough resources to extenuate this problem. Therefore, solely relying on auto-scaling services is not enough to ensure consistent, and exemplary performance of our class of applications. More so, completely relying on auto-scaling services allows for unnecessary over-provisioning in preparation for events such as flash crowds, which is not but at a high cost to the clients.

Multi-cloud, the use of multiple cloud (Grozev and Buyya, 2014) avoids over-provisioning of resources, vendor lock-in, availability, and customisation issues. Multi-cloud deployment has become increasingly popular mainly because of these stated advantages (Grozev and Buyya, 2014). If properly implemented, the multi-cloud deployment model makes it a good fit for overcoming flash crowds and resource failure. Therefore, multi-cloud load balancing is recommended to help avoid overload or per-

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

Adewojo, A. and Bass, J.
Multi-cloud Load Distribution for Three-tier Applications.
DOI: 10.5220/0011092100003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 296-304
ISBN: 978-989-758-570-8; ISSN: 2184-5042

formance degradation caused by resource failure or flash crowd. The key factor in using multi-cloud deployment model to achieve this goal solely relies on the configuration and implementation of this solution, which is the main reason for this research work.

An approach to implement multi-cloud load balancing is to use a centralised load balancer to distribute workload among data centres, such as found in this research (Grozev and Buyya, 2014). Though this approach allows fine-grained control over traffic, it introduces extra latencies to all requests, which reduces the benefit of deploying applications across multi-cloud. However, this approach is suitable when there is a need for legislative control and specific geographic routing of requests.

In this paper, we present a solution that complements and improves the role of auto-scalers for three-tier web-based applications deployed across multi-cloud. We follow the monitor-analyse-plan-execute loop architecture often used by cloud based systems in our proposed solution (Qu et al., 2017). Our proposed solution implements a decentralised approach to multi-cloud geographical load balancing. This ensures consistent high performing web application while maintaining a predefined service level agreement (SLA).

Furthermore, our solution employs a peer-to-peer client-server communication protocol to avoid the overhead incurred by the broadcast protocol used in similar research (Qu et al., 2017). This proposed solution was implemented and evaluated across our experimental test-bed – a heterogeneous combination of one private and two public clouds. We used mainly response times as our determinant metric for evaluating performance.

The key contributions of this research are :

1. a decentralised multi-cloud load balancing architecture that properly distribute the workload of our chosen class of application across multiple clouds;
2. an improved communication protocol of multi-cloud load balancing system; and
3. an implementation and an experimental evaluation of our proposed system using a heterogeneous experimental environment.

The rest of this paper is organised as follows. Section 2 discusses similar research works and how our approach differs to already existing research. Section 3 describes what motivated this research work. We described the multi-cloud deployment model and application requirements in Section 4.1. We introduce and explain our proposed system and its implementation in Section 4. We evaluate our proposed system in Section 5 and presents results in Section 6. Finally, we conclude the paper in Section 7.

2 RELATED WORK

Workload distribution across multi-cloud requires the use of proven and reliable load distribution techniques. There have been various research aimed at distributing workload ranging from popular cloud services to bespoke research services: cloud services such as Amazon Web Service (AWS) Route 53 (Amazon, 2021a), and AWS Elastic Load Balancer (ELB) (Amazon, 2021b); Azure load balancer (Azure, 2021b) and Azure autoscale; overload management (Qu et al., 2017); and geographical load balancing (Grozev and Buyya, 2014).

Cloud services such as ELB load balancer (Amazon, 2021b) can distribute requests to servers in single or multiple data centres using standard load balancing techniques and a set threshold. However, this service can only distribute incoming requests to AWS regions and not third party data centres. Likewise, Azure load balancer (Azure, 2021b) and autoscale (Azure, 2021a) can distribute incoming user requests among servers and data centres owned by Azure alone. These approaches focus on predicting future workloads and provisioning enough resources in advance to accommodate increased workload. The downside of these approaches is that they eventually over provision resources in most cases (Qu et al., 2016; Qu et al., 2017).

Research approaches such as found in (Gandhi et al., 2014; de Paula Junior et al., 2015) reactively provision resources after they detect increased incoming requests or when a set threshold has been met. Furthermore, a similar approach (Qu et al., 2016) proposed the use of spot instances and over-provision of application instances to combat terminations of spot instances and improve workload distribution. However, because resource failures and flash crowds are often unpredictable, it takes the auto-scaler considerable time to provision new resources. Also, it is even more difficult to consistently and evenly distribute load irrespective of an overload or resource failure. Therefore, we argue that it is beneficial to support and improve an auto-scaler to be able to handle situations such as overload and resource failure more effectively.

Researchers (Niu et al., 2015; Javadi et al., 2012) have also used the concept of cloudburst (Ali-Eldin et al., 2014); "the ability to dynamically provision cloud resources to accelerate execution or handle flash crowds when a local facility is saturated," to combat overload and manage increasing user requests.

Grozev (Grozev and Buyya, 2014) proposed an adaptive, geographical, dynamic and reactive resource provisioning and load distribution algorithms to improve response delays without violating legislative and regulatory requirements. This approach dispatches users to cloud data centres using the concept of an entry point of an application framework and a centralised solution.

Qu and Calherios (Qu et al., 2017) adopted a decentralised architecture composed of individual load balancing agents to handle overloads that occur within a data centre by distributing excess incoming requests to cloud data centres with unused capacities. Their approach is composed of individual load balancing agents that communicates using the broadcast protocol to balance extra load. They aimed to complement the role of an auto-scaler, reduce over-provisioning in data centres, and detect short-term overload situations caused by flash crowds and resource failure through the use of geographical load balancing and admission control, so that performance degradation is minimized.

Our approach is different from the above-mentioned approaches. Even though we adopt a decentralised architecture as implemented by (Qu et al., 2017), we do not use load balancing agents, because we want to limit the amount of network broadcast. Furthermore, we argue that we do not need to wait for an overload before distributing requests and, so we aim to consistently distribute workload of cloud deployed web-based three-tier applications instead of combating overloads only. Our framework exemplifies a high availability cloud deployment architecture with peer-to-peer client server communication protocol on an experimental test-bed which comprises three heterogeneous cloud data centres.

3 MOTIVATION AND USE CASE SCENARIOS

The use of multi-cloud can reduce cost and improve resource usage without affecting quality of service (QoS) rendered. In addition, it is common to be able to estimate and plan for traffic spikes, but when the unplanned traffic spikes occur, we need a mechanism to efficiently handle them. Our proposed system improves existing research by (Grozev and Buyya, 2014) and (Qu et al., 2017). It uses the concept of geographical load balancing, dynamic load balancing technique and an improved communication protocol to evenly distribute workload of web application across multi-cloud.

Our solution will be useful for the following scenarios that commonly affect our chosen class of applications:

- Flash Crowds: Flash crowds are unexpected, rapid request surges that commonly occur in web applications (Le et al., 2007; Wang et al., 2011; Ari et al., 2003). They are difficult to manage by only auto-scalers due to their bursty nature. Commercial techniques for handling this scenario is to provision resources after the detection of application overload. Our proposed solution complements auto-scalers by re-distributing requests to available data centres to reduce the occurrence of provisioning new resources and waiting times during resource provisioning when it is necessary to do so.
- Resource Failure : Cloud resource failure is a situation where any of the components in any cloud computing environment experience drastic failure. The three most common resource failures in any cloud environment are hardware, virtual machines, and application failures (Priyadarsini and Arockiam, 2013; Prathiba and Sowvarnica, 2017). Resource failures can happen any time, and can cause performance degradation during resource provisioning if the resource loss is beyond the locally unused resource capacity. Our solution implements a periodic health check to detect all types of failures. Our load balancing service recalculates weights of VM and checks available capacity on a regular basis and if a failure happens before the check, a recalculation is done immediately to properly distribute requests both within the data centre and across all data centres to avoid performance degradation.

4 METHODOLOGY

4.1 Deployment Model and Application Requirements

Our target applications are three-tier web-based business applications across multi-cloud. In addition, to support request forwarding, the application instance in each data centre should be able to communicate with instances deployed in other data centres. We adopt an approach that requires session continuity and data locality to support processing of requests by application replicas deployed across multiple cloud.

Session continuity ensures uninterrupted service experience to the user, regardless of changes to the server or equipment's IP address. Stateless applications, such as search engines and applications that utilises web services to achieve statelessness, does not save client data generated in one session for use in the next session with that client. This and other properties of stateless applications implicitly satisfy the requirement of session continuity.

Data locality ensures that data resides close to the system it supports. In the context of our research, data locality means data should be replicated across multi-cloud, since requests can only be forwarded to data centres with available data. To corroborate this concept for our proposed system, (Grozev and Buyya, 2014) supports data replication for multi-cloud applications because it is a key to good performance (Jacob et al., 2008; Henderson et al., 2015), and thus, improves applicability of our approach.

4.2 Deployment Architecture

We present our decentralised architectural design in Figure 1. This decentralised architectural design features a dynamic load balancing algorithm and technique proposed by (Adewojo and Bass, 2022) and forms part of our multi-cloud load balancing service. We deploy our load balancing service (LBS) as an extra layer of component that augments the three-tier architecture. Each LBS is deployed alongside our application in the same data centre; this helps to reduce latency in detecting workload requests. The services are connected to each other through a virtual private network to ensure communication. Each LBS consist of monitoring, controller, and communication modules. The monitoring module constantly monitors incoming requests and the status of available resources to detect resource failures, increased workload, application, or server workload. The controller module is used to modify the weight of each VM to accommodate request workload. The communication module communicates the capacity and status of each data centre.

4.3 Load Distribution Algorithm

To detect and overcome overload, and resource failures, we use key server metrics of an application server to determine the state of our application servers. The original algorithm by (Adewojo and Bass, 2022) implements a unique weighting technique that combines five carefully selected server metrics utilisation (CPU, Memory, Bandwidth, Network Buffer and thread count) to compute the weight of a VM. Our solution improves the algorithm by including the calculated weight of each data centre that will be used in load distribution and the network latencies between data centres.

To calculate the weight of each data centre, we use the definition of a real-time load $Lr(X_k)$ as described by (Adewojo and Bass, 2022) to calculate the weight of each data centre, as shown in equation (1).

$$W(DC_i) = \frac{\sum \frac{1}{Lr(X_k)}}{n} \tag{1}$$

We abstract our novel multi-cloud load balancing algorithm in Algorithm 1. The first step in the algorithm is to receive and set an overall threshold for the input parameters. The values for these thresholds and how they were calculated can be found in (Adewojo and Bass, 2022). The algorithm loops through a list of VMs and compares each utilisation values against the set threshold. The weight of each VM is then computed and assigned to VMs as described in (Adewojo and Bass, 2022). The algorithm in line 5 further loops through all remote data centres and calculate the weight of each data centre using equation (1). Line 7 assigns the weight of each data centre. If a VM or data centre cannot accommodate any more requests, it sets the weight to zero. The requests are then assigned to servers and data centres based on the assigned weights, as shown in Line 9.

We use the network latency between data centres to determine the nearest data centre to route requests, as shown in line 9 in the algorithm.

The input parameters of the algorithm are:

- Th_c—CPU threshold;
- Th_r—RAM threshold;
- Th_{bw}—Bandwidth threshold;
- Th_{tc}—Thread count threshold;
- VM_{as}—list of currently deployed application server VMs;
- VM_{dc}—list of currently deployed application server VMs per remote data centre;
- $clouds$—list of participating remote data centres;
- L_i—Latency to the ith data centre from the forwarding data centre

4.4 Communication Protocol

We deployed our load balancing solution on each participating data centre. They communicate with each other using a peer-to-peer client-server communication protocol, as depicted in Figure 1. Each solution relays its system state to another solution in a different data centre at a regular predefined time interval

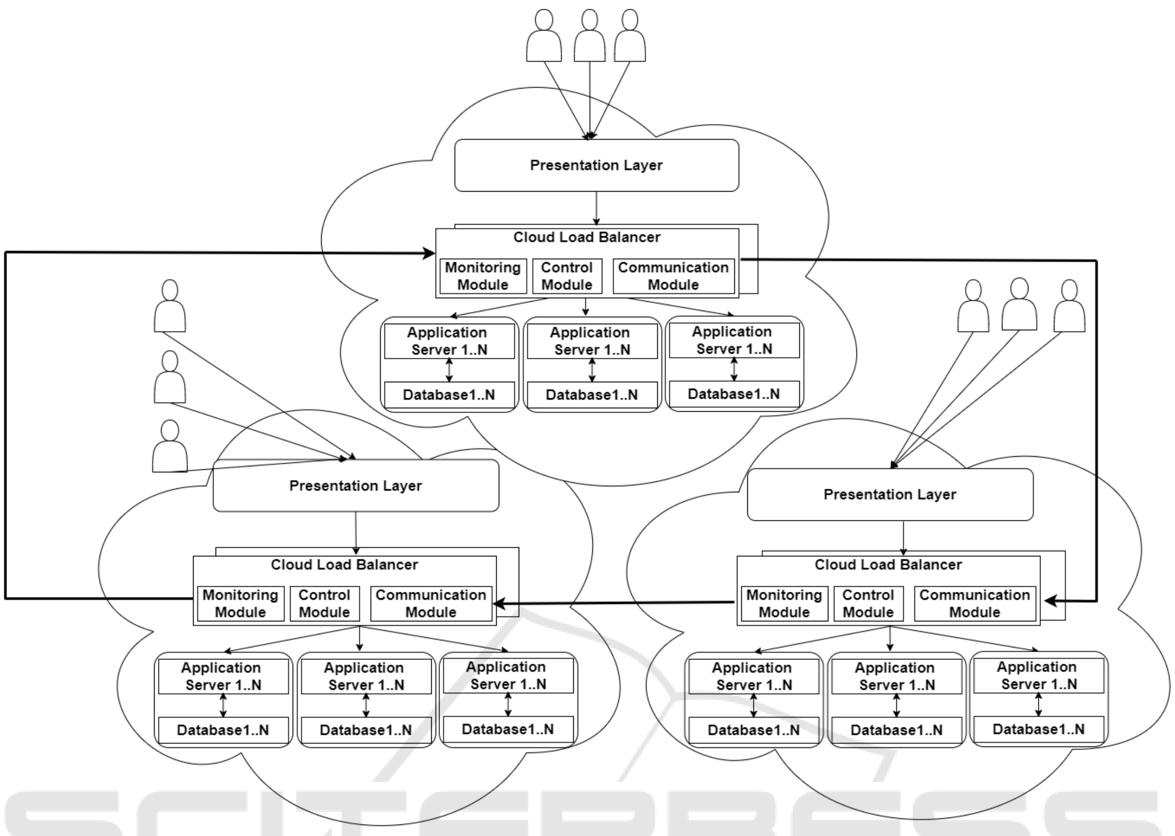


Figure 1: Load Balancing Deployment Architecture.

Algorithm 1: Multi-Cloud Request Handling Algorithm.

```
   Input: $s_i$, $Thc$, $Thr$, $Thbw$, $Th_tr$, $VM_{as}$, $VM_{dc}$, $L_i$
1  RetrieveAllocateToInputAllThresholdValues ();
2  for each VM, $vm_i \in VM_{as}$ do
3      assignweighttoVM $(vm_i, W(X_k))$ according to (Adewojo and Bass, 2022);
4  end
5  for each cloud, $vm_j \in VM_{dc}$ do
6      $W(DC_k) \leftarrow$ CalculateWeightofDataCentre ($Lr_k$ ,$VM_{dc}$, $vm_j$);
7      assignweighttoDC $(vm_{dc}, W(DC_k))$;
8  end
9  HAProxyAssignRequest $(s_i, VM \in clouds, L_i)$
```

of two seconds and every time the load balancer distributes workload. Each communicated system's state always comprises the originated state and the states of the peered system. This chosen mode of communication protocol help to reduce network overhead associated with node communication by only broadcasting to the peered node. It provides significantly better spatial reuse characteristics, irrespective of the number of nodes. As the number of nodes increase significantly, there might be slight degradation in performance, but the advantages definitely outweighs this drawback.

4.5 Algorithm Implementation and Deployment

We implemented our algorithm as a separate program that ties into a state-of-the-art load balancer, HAProxy 2.4.2-1. We created a separate program because HAProxy does not support complex configurations featured in our algorithm. We colocated our program with the HAProxy load balancer to reduce network latency. We used HAProxy's health monitor to monitor the performance indicators and VM's health every 2000ms.

Our program's monitoring module periodically fetches required monitored information using HAProxy's stats application programming interface (API). Then it extracts and manipulates performance values and health statuses of attached VM, and passes them to our control module. The control module activates our algorithm to determine the weight of each VM and data centre. The control module passes the weights to the load balancer and also updates the communication module.

We implement request distribution and admission control by dynamically changing HAProxy's configuration. When required, the control module dynamically creates a new configuration file for HAProxy during runtime. This process automatically reloads the new configuration to the running HAProxy load balancer, then the load balancer distributes requests among the data centre.

To activate request forwarding, each new configuration file contains the IP addresses of the load balancers located in other participating data centres and represents them as normal servers with individual weights. Our program assigns weight to each server. The assigned weight will determine the amount of requests that can be distributed across data centres and VMs in each data centre. HAProxy then uses the weighted round-robin algorithm to distribute requests.

We implement admission control using the Access Control List (ACL) mechanism of HAProxy. We use HAProxy's customised default page to inform users of delay when there is a surge in user requests that consequently affect response times.

5 PERFORMANCE EVALUATION

5.1 Case Study Application

Our case study application is a three-tier stateless E-commerce application that was built using Orchard core framework. We used Elastic search to implement its search engine, the main focus of our experiment. The application consist of a data layer that runs MySQL database loaded with similar products that can be found on eBay; a domain layer that implements buying and selling of products, and a web interface where users can search for products.

5.2 Experimental Test-bed

Our experimental results are the average of 5 repeated experiments over a 24-hour period. Our experimental test-bed consists of 3 heterogeneous data centres; a private cloud running OpenStack, located in London, Amazon Web Service located in Tokyo: ap-northeast-1a, and DigitalOcean located in New York. It is illustrated in Figure 2. Each data centre consists of nine heterogeneous VMs. The private cloud had VMs with 4 and 8 VCPUs, 4GB and 8GB RAM, 40GB and 80GB disk size. AWS had VMs with 2 VCPUs, 4GB and 8GB RAM, and 20GB disk size. DigitalOCean had VMs with 2 VCPUs, 4GB RAM and 80GB disk size. We measured and recorded the Round-trip Time (RTT) latencies between the data centres using ping. The RTT are: London-Tokyo-London : 1.68ms and London-New York-London: 240.53ms.

In each data centre, we deployed HAProxy server along with our load balancing algorithm on two VMs; one VM acts as a standby, depicting a high availability architecture. We deployed our application servers on five VMs and database servers on two VMs. Furthermore, we deploy a standard auto-scaler on each data centre. In order to simulate real user request and location, we deployed Apache Jmeter (our workload simulator) on an external standalone machine with 4-core, Intel Core i7, 2.8GHz CPU and 8Gigabit Ethernet NIC.

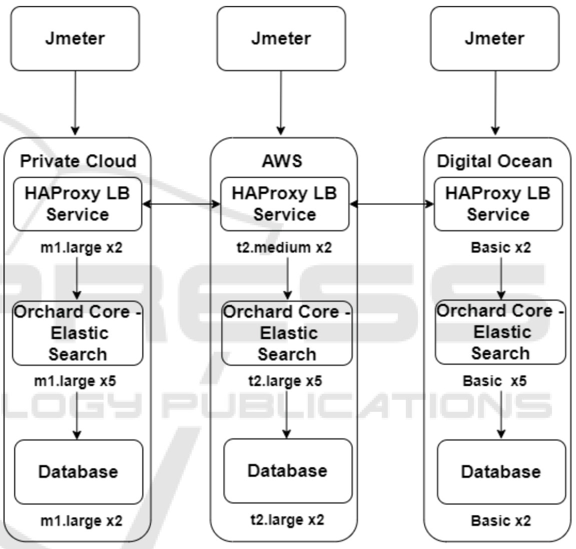


Figure 2: Experimental Test-bed.

5.3 Workload

To implement our profiling test, we sent e-commerce search requests using Jmeter to our cloud deployed applications. Firstly, we stipulated that 90% of requests should be replied within 1 second. Secondly, we performed tests to determine the average requests that each class of our application servers can handle without violating the SLA. We created workloads using the proposed workload model by (Bahga et al., 2011).

Based on this workload model, we created three workloads for the three data centres using parameters stated in Table 1. The average of the largest amount of requests that can be handled by the application servers are 80 (private cloud), 65 (DigitalOcean) and 45 (AWS) requests/s.

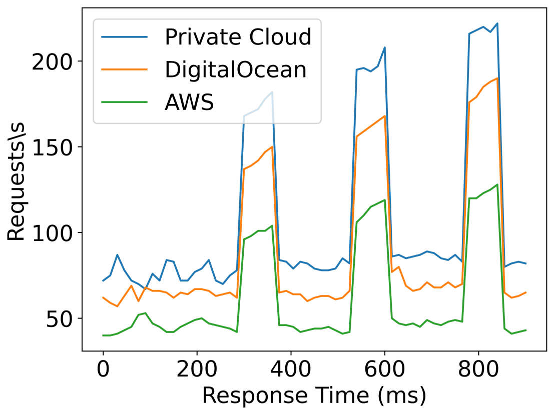


Figure 3: Experimental Workloads with flash crowds ranging from 110% to 190% of the normal load.

To simulate flash crowd, we created two extra workloads with increased requests, as shown in Figure 3. Each workload experiences a total of three seconds flash crowds within a period of 1 minute. The peak of the flash crowd range from 110% to 190% of the normal workload. The experiment experiences flash crowds starting from 300ms time point in any time frame.

To test our approach when there is resource failure, we ramped up average incoming requests to 240 requests/s, representing the highest bound of our normal workload. Starting from 300ms, we simulate resource failure that lasts for 300ms, this also experiences a total of three seconds resource failure within 1 minute interval.

5.4 Benchmarks

To validate and compare the performance of our solution, we benchmark our results with the following:

- Request Queuing: This benchmark process queues up all requests in the local servers, imposes no admission control, does no geographical balancing, and uses just the round-robin algorithm. This imitates the situation that an autoscaler is booting a new VM within a data centre.
- Admission control: This benchmark process directly imposes admission control when distributing requests. It lets the load balancer redirect requests at first and if there is no capacity to accept the redirected requests, it sends a message to users to tell them they are in a queue.

6 RESULTS

6.1 Resource Failures

To test resource failures, we removed some VMs from the load balancer pool at 300ms time point and added them back to the pool after 5 seconds to imitate recovery from failure. We repeated this experiment for each of the data centres such that we simulated resource failure for each data centre. We also conducted more experiments where resource failures occurred in combinations of the data centres.

Figure 4 showed performance of the system during one server failure. It showed that without our approach, all data centres would not maintain the defined SLA. Furthermore, the other approaches exhibited higher response times, which indicated performance degradation. This same characteristics were exhibited in two server failures scenarios; we performed the test on a combination of all participating clouds. They all could not attend to 90% of requests at a lesser response times compared to our approach.

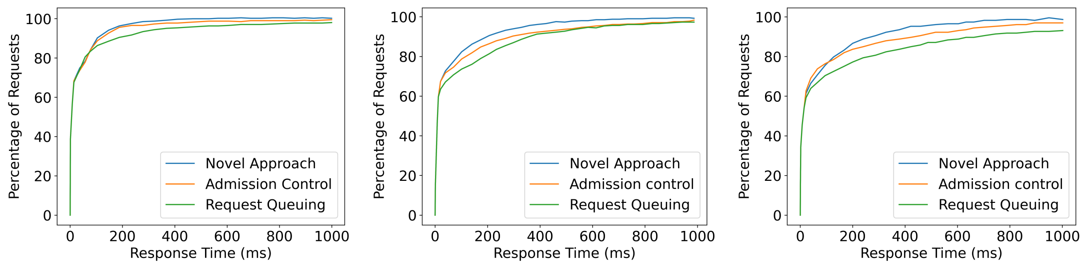
Figure 5 shows the performance of our algorithm and the benchmarks when there were three VM failures. This approach made the data centres become unresponsive, unlike our novel approach that was still able to maintain defined SLA even though the response time was high. In summation, our approach outperformed the response times of both admission control and request queuing benchmark by 6.7%. This means our approach can handle more workload with an acceptable response time during server failure scenarios.

6.2 Flash Crowds

We tested our approach by simulating flash crowds in each of the data centres. Figure 6 shows how our approach and benchmarks performed under flash crowds. Our experiments showed that our approach outperformed our benchmarks at every instance of flash crowds. We recorded an improvement in the percentage of requests handled. Our approach improved response times by 4.08% and 20.05% relatively to admission control and request queuing benchmarks, respectively. This confirms that our solution can consistently distribute the request of our class of applications even during flash crowds. We note that the size of the VM also determines the performance, we believe a better optimised VM for web applications will offer a lesser response times if it is coupled with our solution.

Table 1: Workload Parameter.

	Mean	Min	Max	Deviation
ThinkTime	4000	100	20000	2
Intersession Interval	3000	100	15000	2
Session Length	10	5	50	2



(a) 1 Server Failure in Private Cloud. (b) 1 Server Failure in DigitalOcean. (c) 1 Server Failure in AWS.

Figure 4: Cumulative Distribution Values of One Server Failure.

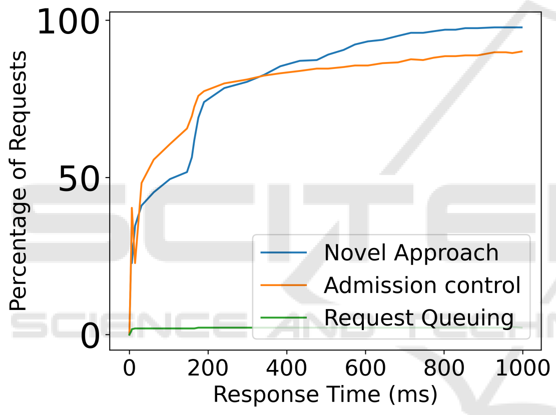


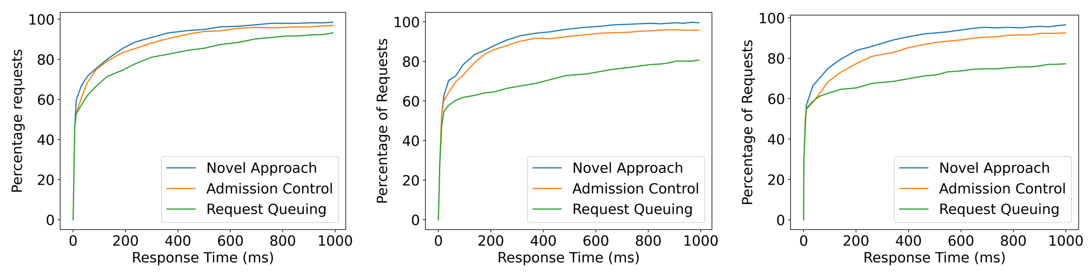
Figure 5: Three Server Failures.

7 CONCLUSIONS AND FUTURE WORK

Cloud deployed web-based applications commonly experience flash crowds that might result in resource failure and/or performance degradation. To resolve this problem, we proposed a multi-cloud decentralised load balancing system.This system effectively distributes the workload of this class of applications using geographical dynamic load balancing to minimise performance degradation and improve response time. Our approach deployed our load balancing solution in each data centre for quick sensing of overload and resource failure occurrence. Our load balancing solution comprises HAProxy and an improved novel load balancing algorithm (that utilises five carefully selected server metrics to determine the real-time load of VMs) to include multi-cloud weighting and request distribution.

We implemented and evaluated our algorithm across a private cloud located in London running OpenStack, AWS located in Asia data centre and DigitalOcean in US data centre. We validated our algorithm by comparing it to two benchmarks; request queuing and standard admission control methods. To test the applicability of our solution, we simulated flash crowds and resource failures using our experimental tools to send requests spikes and remove VMs, respectively. We carefully measured response times of our experiments and obtained results showed that our approach maintained accepted SLA of requests during flash crowds and resource failure. Furthermore, it improved response times performance by 6.7% during resource contention periods and 4.08% and 20.05% during flash crowd scenarios when compared with admission control and request queuing, respectively. This validates that our proposed approach improves the performance of multi-cloud deployed web-based three-tier application and effectively distributes the workload of these applications.

In future, we hope to tackle some limitations of this research. We will consider using domain specific languages such as Cloud Application Modelling and Execution Language (CAMEL) to describe our deployment approach. We also will compare our approaches with some popular approaches such as the use of serverless technologies.



(a) 140 req/s flash crowd. (b) 190 req/s flash crowd. (c) 240 req/s flash crowd.

Figure 6: Cumulative Distribution Values of Flash Crowds using Different Approaches.

REFERENCES

Adewojo, A, A. and Bass, M, J. (2022). A novel weight-assignment load balancing algorithm for cloud applications. In *12th International Conference on Cloud Computing and Services Science*, page TBD. IEEE.

Ali-Eldin, A., Seleznjev, O., Sjöstedt-de Luna, S., Tordsson, J., and Elmroth, E. (2014). Measuring cloud workload burstiness. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 566–572. IEEE.

Amazon (2021a). Amazon route 53.

Amazon (2021b). Elastic load balancing.

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.

Azure, M. (2021a). Azure autoscale — microsoft azure.

Azure, M. (2021b). Load balancer documentation.

Bahga, A., Madisetti, V. K., et al. (2011). Synthetic workload generation for cloud computing applications. *Journal of Software Engineering and Applications*, 4(07):396.

de Paula Junior, U., Drummond, L. M., de Oliveira, D., Frota, Y., and Barbosa, V. C. (2015). Handling flash-crowd events to improve the performance of web applications. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 769–774.

Gandhi, A., Dube, P., Karve, A., Kochut, A., and Zhang, L. (2014). Adaptive, model-driven autoscaling for cloud applications. In *11th International Conference on Autonomic Computing ({ICAC} 14)*, pages 57–64.

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

Henderson, T., Michalakes, J., Gokhale, I., and Jha, A. (2015). Chapter 2 - numerical weather prediction optimization. In Reinders, J. and Jeffers, J., editors, *High Performance Parallelism Pearls*, pages 7–23. Morgan Kaufmann, Boston.

Jacob, B., Ng, S. W., and Wang, D. T. (2008). Chapter 3 - management of cache contents. In Jacob, B., Ng, S. W., and Wang, D. T., editors, *Memory Systems*, pages 117–216. Morgan Kaufmann, San Francisco.

Javadi, B., Abawajy, J., and Buyya, R. (2012). Failure-aware resource provisioning for hybrid cloud infrastructure. *Journal of parallel and distributed computing*, 72(10):1318–1331.

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.

Niu, Y., Luo, B., Liu, F., Liu, J., and Li, B. (2015). When hybrid cloud meets flash crowd: Towards cost-effective service provisioning. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1044–1052. 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. (2016). A reliable and cost-efficient auto-scaling system for web applications using heterogeneous spot instances. *Journal of Network and Computer Applications*, 65:167–180.

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.

Wang, J., Phan, R. C.-W., Whitley, J. N., and Parish, D. J. (2011). Ddos attacks traffic and flash crowds traffic simulation with a hardware test center platform. In *2011 World Congress on Internet Security (WorldCIS-2011)*, pages 15–20. IEEE.