# Using Multithreaded Load Balancer to Improve Connection Performance in Container Environment

**Pang-Wei Tsai**
*Department of Information Management, National Central University, No. 300, Zhongda Rd., Zhongli Dist. Taoyuan City 320, Taiwan.*

**Hong-Yu Wei**
*Department of Information Management, National Central University, No. 300, Zhongda Rd., Zhongli Dist. Taoyuan City 320, Taiwan.*

**Yu-Chi Hsu**
*Department of Information Management, National Central University, No. 300, Zhongda Rd., Zhongli Dist. Taoyuan City 320, Taiwan.*

*E-mail: pwtsai@ncu.edu.tw, 109423069@cc.ncu.edu.tw, yuchi.hsu@g.ncu.edu.tw*
*www.ncu.edu.tw*

**Abstract**

The virtualization technology has been widely used in cloud service for a long period of time, it provides resource allocation, flexibility and efficiency by using virtual machine. Nevertheless, for lightweight workload, using container gains more advantages on transferability, rapid deployment, and robust ecosystem supports. Because of such characteristics, container is commonly used in edge computing and micro-service to fulfill essential resource requirements. However, when service deployed on the container receives too much requests without limitation, it may meet degrading effect which lowers down its performance and makes user experience to be poor. Hence, this research aims to investigate container request-response issue and implement multithreaded load balancer to improve HTTP connection performance equally for web-based application service.

*Keywords*: Virtualization, Container, Multithreaded, Load Balancer.

## 1. Introduction

With the growth of internet, requirements of networking service and application such as on-demand, quality assurance, and user experience are getting more and more important [1]. To provide stable, smooth, and persist data exchanges, how to design and develop a scalable and efficient system framework is a challenge [2]. For infrastructure provider, finding suitable solutions to support upper-layer service and application gains more benefits in production operation. Furthermore, the budget is an important thing as well. Not only hardware but software take cost in both deployment and maintenance. Therefore, it is difficult to find a fixed strategy to fulfill requirements of every applications and services for approaching cost-efficiency and high-performance.

Generally, user may access the application service from time to time. Depending on user amount, the resource consumption during rush and off-peak hours could be very different, especially for web-based services [3]. Hence, web service providers might be in doubt about paying more money to prepare extra resources to support uncertain flooding connections in a short time. Fortunately, the virtualization technology provides more flexibility and adaptation in infrastructure deployment. Considering the used computing component for running

service software could be a physical server or virtual machine, enabling a control mechanism to allocate adequate resource to fulfill the requirement is necessary. Such resource on-demand concepts [4] has been widely used in cloud-based operation models. By doing this, service providers are able to proceed their preferred business models with performance-priority, essential low-cost, resilience or other strategies.

For most cloud resources, bare-metal, virtual dedicated, and virtual private servers are common units in selection. To allocate computing and networking resources with cost-efficiently, many researches [5][6] focus on auto-scaling and migration mechanisms to gracefully manage both spending and performance in balance. However, for lightweight micro-services, the overcapacity resource of virtual machine is too much. Hence, container solutions like Docker [7] has become a solution in supporting such operation scenes during deployment.

To improve the performance of lightweight web-based services running on container environment, this paper presents an investigation and uses multithreaded design to dispatch incoming connection to the pod in Kubernetes [8] environment. It aims to make adequate allocation for satisfying user HTTP requests to allocated pods.

## 2. Backgrounds and Related Work

### 2.1. *Container*

Even though the virtual machine solves the problem of isolation and security in sharing host resource [9], to prepare a production environment, operator still requires to treat virtual machine as a physical one (like installing OS). The hardware-level of virtual machine may gain more benefits from virtualization, while the OS-level is not exactly [10]. The resource of virtual machine might be too much for running some micro-services and their corresponding applications. Due to above reasons, the container starts to substitute traditional virtual machine for handling such lightweight tasks in recent years [11]. While there are still several issues about using container, such as software dependency and limited capability.

### 2.2. *Dockerized Environment*

For enabling container environment, using Docker framework [7] is a common choice. There is no longer need for cloud service provider to setup host environment for running application services separately. The container image could be prepared initially to support scale-up and flexible deployment in production use. With Kubernetes, the control mechanism is able to supervise created containers transparently [8]. Since created containers with the same purpose can be grouped as a pod, operators are able to monitor workload and network for making adaptation adjustment to satisfy user requirements well.

### 2.3. *Discussion*

According to the study made by S. K. Mohanty et al. [12], in different open-source container system architecture, the response latency of during the bursty has meet the bottleneck due to the in-efficient dispatching. Meanwhile, J. Li et al. [13] also compares several (e.g., Knative, Kubeless, OpenFaaS) frameworks and traces their API communication interactions. Both two researches indicate how to dispatch ingress flows to available pod is the key-point to make load balance in connection route. Hence, this research aims to investigate above issues, trying finding an equal strategy to gain more efficiency for user HTTP request-response.

## 3. System Design

To be capable of dispatching incoming request to assigned pod in Kubernetes, authors of this paper have modified the packet handling process of Kubernetes environment. The designed system architecture is shown as Fig. 1, and the added components are listed below:

- **Flow Controller:** It is the supervisor used to manage incoming request and determine forwarding action. The rule will be updated to packet filter and router in Kubernetes. It also keeps monitoring loading stats of each pod to approach balancing control.
- **Ingress Setup Module:** This module aims to receive incoming request and clarify the target path.
- **Pod Allocation Module:** This module maintains the available pod list for making forwarding decision.

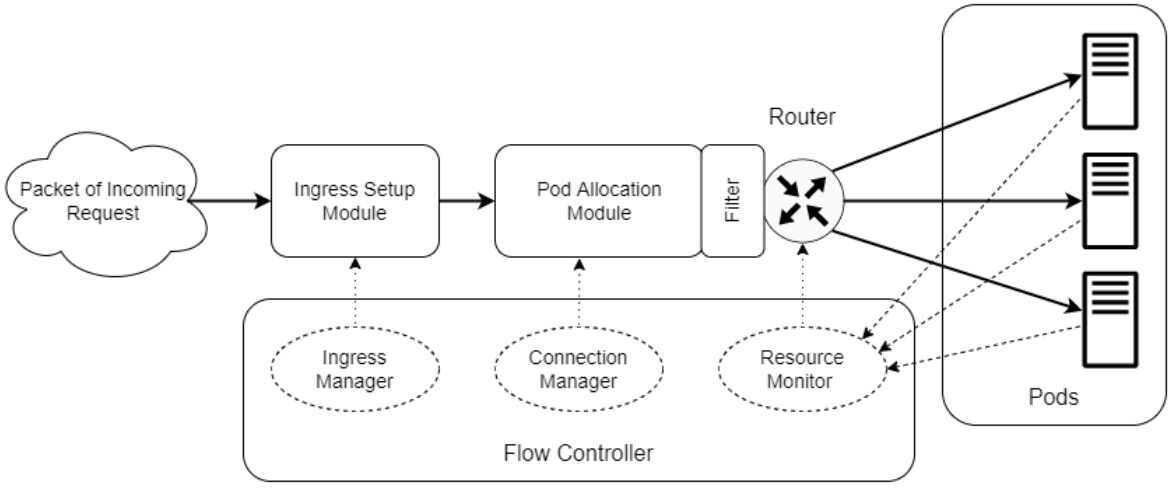


Fig. 1. The modified connection route architecture in Kubernetes.

- **Pod(s):** The component responds user requests. Each pod can be consisted of one or more containers.

For every incoming request, the packet header will be used to make determination. The new request is going to be investigated by flow controller. After completing ingress setup and finding available pod, the request will be guided to the assigned pod through the established socket. All actions are supervised by the flow controller module, and its resource monitor keeps querying pod stats to understand current loading.

### 3.1. *Multithreaded Balancer*

During the operation, the balancing thread is expected to handover the request to flow control process, and the request will be guided by new forked load balancing thread. By doing this, it saves the waiting time for completing overall connection. Once the resource monitor finds any pod is getting overload or becoming malfunctioned, the flow controller should alter the routing strategy or ask physical host to create more pods for sharing the request.

### 3.2. *Request Allocation Algorithm*

The developed algorithm for allocation is shown below:

**Algorithm I: Connection Dispatcher**

R: Request from user
$P_h$: Packet Header
$P_f$: TCP flag of Packet Header
$P_u$: URL in HTTP request
$T_p$: Target Pod in available Pod List

```
1:  if Ph ∈{TCP connection} then {
2:    if (Pf == FIN) then
3:      return termination (R, FIN-ACK)
4:    elseif (Pf == SYN) then
5:      return response (R, SYN-ACK)
6:    elseif (Pf == ACK) then
7:      return response (R, NULL)
8:  }
9:  else {
10:   if Ph ∈{HTTP connection} then {
11:     Pu = fetch_route (R)
12:     if (Pu != NULL ) then {
13:       Tp = dispatch (R)
14:       return allocate_request (R)
15:     }
16:   }
17:   else
18:     return response (R, NULL)
19: }
```

### 3.3. *Balancing Strategy*

For making simple proof-of-concept, the applied strategy in development is using equal-cost policy to determine which pod should serve the new incoming request. According to monitoring data, flow controller will take consideration with ingress path, IP address, request URL, available pods (including workload and memory utilization) to decide which pod should be selected.

Hence, any pod with lowest utilization has priority chance to be allocated. In addition, although the default policy is trying to approach equality, it is still possible to change to be the weighted one for fulfilling purposes in different operation scenarios.

## 4. Experiment and Evaluation

To make evaluation, the experiment environment was conducted by one client and one hosting PCs (with Intel I7-10700 CPU, 64GB DDR4 RAM, WD NVME SSD, and Intel 82576 Network Adapter). The developed modules aim to enable load balancing mechanisms to dispatch the connection sent from client PC (emulated by JMeter [14] - a stress test toolkit) to the assigned pod.

### 4.1. *Experiment Design*

The used Docker template in experiment involves Ubuntu 18.04 and pre-configured Apache2. There are 5 customized HTTP documents placed in folders. The designed load balancing mechanism will guide each request to a small scale resource pool with 25 pods equally. To determine whether the request-response action made by JMeter is success, based on a technical report announced by Google [15], authors decided to set 3 seconds as the maximum waiting period of threshold.

### 4.2. *Evaluation Results*

In experiment, the number of JMeter request was started at 100 (and added 100 in batch) randomly. According to the experiment result, the system is capable of satisfying about 600-700 requests in 1 minute with 25 pods. Over this value, the waiting time will be increased rapidly and determined as failed request-response interaction (see Fig. 2 and Fig. 3). To understand what will be the changes when increasing more pods, authors also created additional pod during 700 requests. While it seems like the bottleneck is not load balancing module because of the number of failed interaction was lowering down when changing from 25 to 50 pods in experiment.

## 5. Conclusions

This research proposed a multithreaded load balancing development in container environment. By using JMeter to evaluate the success rate of request-response interaction, the experiment results show that the multithreaded balancer is able to dispatch hundreds of connections equally with pre-defined mechanism.

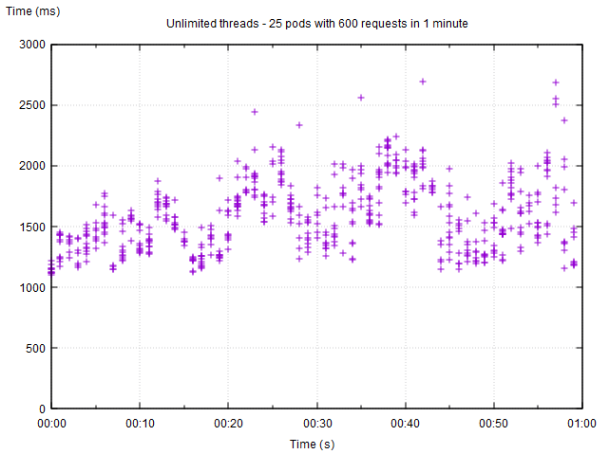


Fig. 2. The JMeter stress test results of random 600 requests in 1 minutes with 25 pods.

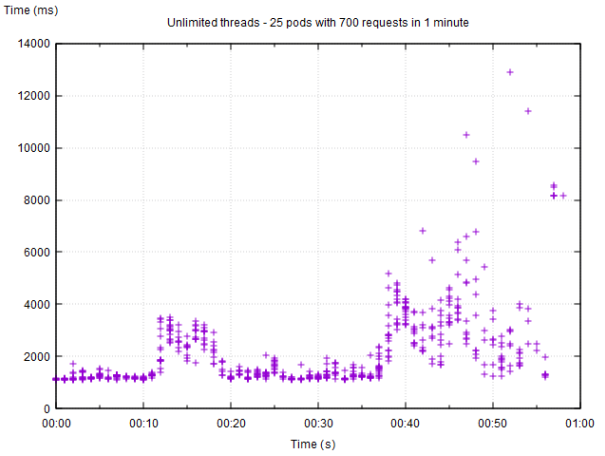


Fig. 3. The JMeter stress test results of random 700 requests in 1 minutes with 25 pods.

**Acknowledgments**

This research was supported in part by NSC of Taiwan (109-2222-E-008-005-MY3 and 111-2218-E-006-010-MBK). Authors are also grateful to TWAREN SDN research teams for their great help.

## References

1. T. Zhao et al., “QoE in video transmission: A user experience-driven strategy,” IEEE Communications Surveys & Tutorials, vol. 19, no. 1, pp. 285-302, 2016.

2. S. Lehrig et al., “Scalability, elasticity, and efficiency in cloud computing: A systematic literature review of definitions and metrics,” in proceedings of the international ACM SIGSOFT conference on quality of software architectures, pp. 83-92, 2015.
3. A. L. Lemos et al., “Web service composition: a survey of techniques and tools,” ACM Computing Surveys vol.48, no.3, pp. 1-41, 2015.
4. Y. Tokusashi et al., “The case for in-network computing on demand,” in proceedings of the EuroSys conference, pp. 1-16, 2019.
5. E. Roloff et al., “High performance computing in the cloud: Deployment, performance and cost efficiency,” in proceedings of the IEEE International Conference on Cloud Computing Technology and Science Proceedings, pp. 371-378, 2012.
6. K.R.R. Babu and P. Samuel, “Interference aware prediction mechanism for auto scaling in cloud,” Computers & Electrical Engineering, vol. 69, pp. 351-363, 2018.
7. D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.
8. B. Burns et al., “Kubernetes: up and running,” O'Reilly Media, Inc., 2022.
9. A. V. Cleeff et al., “Security implications of virtualization: A literature study,” in proceedings of the international conference on computational science and engineering, pp. 353-358, 2009.
10. J. Sahoo et al., “Virtualization: A survey on concepts, taxonomy and associated security issues,” in proceedings of the international conference on computer and network technology. pp. 222-226, 2010.
11. I. Baldini et al., “Serverless computing: Current trends and open problems,” Research Advances in Cloud Computing, Springer, pp. 1-20, 2017.
12. S. K. Mohanty et al., “An evaluation of open source serverless computing frameworks,” in proceedings of the IEEE International Conference on Cloud Computing Technology and Science, pp.115-120, 2018.
13. J. Li et al., “Analyzing open-source serverless platforms: characteristics and performance,” arXiv:2106.03601, pp. 1-7, 2021.
14. B. Erinle., “Performance testing with JMeter,” Packt Publishing Ltd, 2015.
15. D. An, “Find out how you stack up to new industry benchmarks for mobile page speed,” [Online]. Available: https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/. [Dec. 10 2022].


## Authors Introduction


Dr. Pang-Wei Tsai

He received the B.S. degree in Electrical Engineering and the M.S. / Ph.D. degrees in Computer and Communication Engineering from National Cheng Kung University. His research interests include SDN, cloud computing, information security, and network testbed.

Mr. Hong-Yu Wei

He received the M.S. degree in Department of Information Management from National Central University, 2022. His research interests include Internet technology, cloud application, and Kubernetes.

Mr. Yu-Chi Hsu

He received the B.S. degree in Information Management in 2019 form the Fu Jen Catholic University. He is currently studying in National Central University for M.S. degree. His research interests are Internet technology, network management, and information security.