Adding a Dynamic Load Balancing based on a Static method in Cloud via OpenStack

by

Neda Lame

THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
IN PARTIAL FULFILLMENT OF A MASTER'S DEGREE
WITH THESIS IN IT ENGINEERING
M.A.Sc.

MONTREAL, AUGUST 01, 2023

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Michel Kadoch, Thesis supervisor
Département de génie électrique, École de technologie supérieure

Mr. Alain April, Chair, Board of Examiners
Département de génie logiciel et des TI, École de technologie supérieure

Mr. David Bensoussan, Member of the Jury
Département de génie électrique , École de technologie supérieure

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON JULY 25, 2023

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Prof.Kadoch, for his support, guidance, feedback, and encouragement throughout the research process. Thank you for all your constructive advice and consistent support during the running of this project.

Also, I would like to extend my deepest gratitude to my parents for their love, support, and understanding during this challenging time that I am away from them to reach my educational goals. Their encouragement gave me the strength and motivation to pass difficult moments successfully.

Finally, I would like to thank my friends for their help and support in my academic journey and for generously sharing their time, knowledge, and experiences.

Ajout d'un équilibrage de charge dynamique basé sur une méthode Statique dans le Cloud via OpenStack

Neda Lame

RÉSUMÉ

Avec l'émergence de la technologie informatique en nuage sur Internet, l'accès à l'informatique partagée ressources à bas coût par les consommateurs augmente. En raison de la demande massive de services en ligne services, l'équilibrage de charge est essentiel pour optimiser l'utilisation des ressources et éviter la surcharge ou sous-charge résultant des mises à jour du système d'exploitation, du temps d'exécution des tâches, de la panne du serveur sur les fournisseurs et une défaillance du système due à des problèmes matériels. Pour relever ces défis de l'équilibrage de charge dans le cloud computing, divers algorithmes d'équilibrage de charge ont été étudiés dans modes statique et dynamique. Bien que des mécanismes d'équilibrage de charge dynamique aient été proposés avec leurs avantages et leurs inconvénients, aucun n'a été déployé sur la base des méthodes statiques actuelles dans la plate-forme OpenStack, qui est gratuite et dispose d'une communauté géante. Par conséquent, nous apporter une solution dynamique basée sur la méthode statique, Weighted Round Robin, dans OpenStack. Notre solution dynamique est capable d'équilibrer les tâches en tenant compte des critères d'utilisation du processeur. En tant que évaluation, nous avons analysé les performances de la méthode statique et de notre solution en tant que dynamique version de la méthode standard. Nos résultats expérimentaux ont montré un temps de traitement moyen inférieur et de meilleures performances dans notre solution que la méthode statique. Comme travail futur, nous avons proposé trois méthodes suivantes. La première consiste à prendre en compte plusieurs dimensions dans notre solution pour équilibrage de charge dynamique dans OpenStack pour améliorer l'efficacité. Ensuite, utilisez plusieurs nuages dans différentes régions géographiques pour équilibrer la charge de travail entre les fournisseurs de ces régions. Le dernier est une autre conception de notre solution dynamique basée sur la méthode statique, Weighted Round Robin dans OpenStack à l'aide de Machine Learning, avec suivi des charges actuelles, du processeur, de la mémoire et Réseau pour calculer les pondérations appropriées pour chaque serveur principal afin d'équilibrer les demandes des utilisateurs. En outre, la mise à l'échelle des ressources du cluster de serveurs peut être utilisée via Machine Learning pour provisionner une nouvelle machine virtuelle dans une période mouvementée pour répondre aux demandes des clients.

Mots-clés: Cloud computing, Équilibreur de charge, Équilibrage de charge, Équilibrage de charge statique, Équilibrage de charge dynamique, OpenStack

Adding a Dynamic Load Balancing based on a Static method in Cloud via OpenStack

Neda Lame

ABSTRACT

With emerging Cloud computing technology over the Internet, accessing shared computing resources at low cost by consumers is increasing. Due to the massive demand for online services, load balancing is essential to optimize resource utilization and prevent overloading or underloading resulting from operating system updates, task operating time, server failure on the providers, and system failure due to hardware issues. To address these challenges of load balancing in cloud computing, various load balancing algorithms have been studied in static and dynamic modes. Although dynamic load balancing mechanisms have been proposed with their merits and demerits, none have been deployed based on the current static methods in the OpenStack platform, which is free of cost and has a giant community. Therefore, we contribute a dynamic solution based on the static method, Weighted Round Robin, in OpenStack. Our dynamic solution is able to balance the tasks considering CPU utilization criteria. As an evaluation, we analyzed the performance of the static method and our solution as a dynamic version of the standard method. Our experimental results showed less mean processing time and better performance in our solution than the static method. As a future work, we proposed three following methods. The first is considering multiple dimensions within our solution for dynamic load balancing in OpenStack to enhance efficiency. Next is using multiple clouds in different geographical regions to balance the workload among the providers in those regions. Last is a further design of our dynamic solution based on the static method, Weighted Round Robin in OpenStack using Machine Learning, with tracking current loads, CPU, Memory, and Network to calculate the suitable weights for each backend server to balance the users' requests. Also, scaling out the server cluster's resources can be employed through Machine Learning to provision a new virtual machine in a hectic time to serve the clients' requests.

Keywords: Cloud computing, Load balancer, Load balancing, Static load balancing, Dynamic load balancing, OpenStack

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABBREVIATIONS

API	Ápplication Programming Interface
AWLLB	Adaptive Weighted Least Load Balancing algorithm
A2LB	Autonomous Agent-based Load Balancing algorithm
DWRR	Dynamic Weighted Round Robin
FwaaS	Firewall as a Service
FCFS	First Come First Serve
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a Service
ICT	Information and Communication Technology
LBaaS	National Institute of Standards and Technology
PaaS	Platform as a Service
RR	Round Robin
SDN	Software Defined Networking
SLA	Service Level Agreements
SaaS	Software as a Service
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WLC	Weighted least connection
WS	Web Server
WRR	Weighted Round Robin

INTRODUCTION

Nowadays, the use of Cloud Computing services is growing in Information and Communication Technology (ICT) instead of establishing physical data centers due to reducing the operating expenses and time required for providing online services by cloud computing providers. The online services consist of network resources to serve the end users' requirements through different cloud providers. Moreover, clients need to access their online services with vast requests. In this case, it is necessary to use a load balancer to distribute incoming and outcoming traffic between servers or clients' requests. In cloud computing, load balancing is also called LBaaS, Load Balancing as a Service (Mishra, Sahoo & Parida, 2020).

There are different load balancers, hardware and software-based. However, the hardware load balancer is more expensive and harder to configure than the software one. Therefore, using a software load balancer in cloud computing, which is part of SDN, Software Defined Networking, is a solution to distribute the resources dynamically (Al-Mashhadi *et al.*, 2020).

In general, load balancing is divided into both static and dynamic techniques. The static methods work based on initial knowledge of servers and assign the workload such as CPU, Network and Storage capacity among servers equivalently. However, the dynamic techniques distribute the loads based on the last status of servers at run-time. Although these dynamic algorithms are complex, they have fault tolerance and better performance (Kumar & Rana, 2015). Moreover, (Singh, Juneja & Malhotra, 2015) stated that dynamic load balancing methods concentrate on low latency and execution time and are appropriate for large-scale distributed environments. In contrast, static ones are good for small-scale distributed environments with high-speed Internet regardless of latency.

The software-based load balancer in cloud computing can be deployed on different environments, such as OpenStack, an open-source software cloud computing platform. OpenStack includes a virtual network framework based on SDN architecture to provide and manage the network

environment and virtual network infrastructure (Rista, Ajdari & Zenuni, 2020). Using SDN in cloud computing to deploy a dynamic load balancing reduces operation and capital costs (Kumar & Rana, 2015). However, OpenStack uses static methods for load balancing by default without considering the CPU usage of each VM (OpenStack, 2018a). Therefore, load balancing in OpenStack can not be dynamically done based on a common popular static method at real-time.

The main goal of our study is to add a dynamic version of static load balancing, Weighted Round Robin (WRR)[1], in the Cloud via OpenStack to optimize the loads. Our approach is to add WRR in dynamic state. The modified WRR uses the last status of servers in a network by considering a standard dimension, such as CPU usage, to accommodate the upcoming request for each server. Our design DWRR[2] assigns workload across the servers at run-time. Then, our solution is compared to the standard method, Weighted Round Robin, on OpenStack in terms of minimum response time and maximum hardware utilization.

In this study, we implemented an agent-based load balancer based on a metric such as CPU utilization to balance the loads dynamically. Our experimental results showed that the CPU usage metric resulted in minimum response time and maximum resource utilization. The experiment is performed in a cloud environment via OpenStack on VirtualBox[3] hypervisor to emulate a cloud network with all virtualized components to implement the proposed solution.

In the proposed algorithm (DWRR), two threads are considered written by Python. The main thread on the load balancer is in charge of monitoring and processing the information coming from servers. The worker thread on servers is in charge of periodically updating each server's numeric weight[4] value via the load balancer.

---

[1] WRR is Weighted Round Robin, a static load balancing method which is in default used in OpenStack at present.

[2] DWRR is a dynamic version of WRR, Weighed Round Robin, designed for our study.

[3] VirtualBox is a open-source virtualization software for creating virtual environment.

[4] Weight is selected based on power usage in which maximum weight will be assigned with maximum power usage and minimum weight will be assigned with minimum CPU usage.

In conclusion, we successfully implemented a dynamic version of Weighted Round Robin (DWRR) on OpenStack, which is also compared with the static version (WRR). Furthermore, we observe that by changing the static load balancing to a dynamic state in OpenStack, we reach a small percentage of improvement in efficiency for processing tasks and time execution.

This thesis is divided into the following chapters: The Introduction consists of the thesis objectives.

Chapter 1 covers the background concepts behind cloud computing, SDN technology, load balancing in the Cloud and OpenStack components.

Chapter 2 describes the related works discussed regarding other load balancing techniques in the cloud environment.

Chapter 3 includes the implementation of the proposed dynamic load balancing method and installation environment.

Chapter 4 comprises analyzing the results of experiments and negotiation based on them.

Chapter 5 concludes with the outcomes and defines future work to improve the efficiency of the dynamic solution in OpenStack.

CHAPTER 1

BACKGROUND

We are going to present the basic concepts and the terminology of Cloud computing, SDN, OpenStack architecture and essential modules, and load balancing techniques to follow the results presented in this thesis. Furthermore, we will describe the core terms related to load balancing in OpenStack.

1.1 Cloud Definition

"Cloud is a parallel and distributed computing system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service level agreements (SLA) established through negotiation between the service provider and consumers."(Buyya, Yeo, Venugopal, Broberg & Brandic, 2009)

In simple words, Cloud is network access to shared resources and standardized by NIST (National Institute of Standards and Technology). Cloud Computing has an official definition by NIST, which is "a pay-per-use model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell & Grance, 2011).

According to the definition of NIST, cloud computing has at least five characteristics, including on-demand self-service, measured service, resource pooling, broad network access, and rapid elasticity. It is also possible to receive three service models consisting of Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). Figure 1.1 shows the details regarding service models.

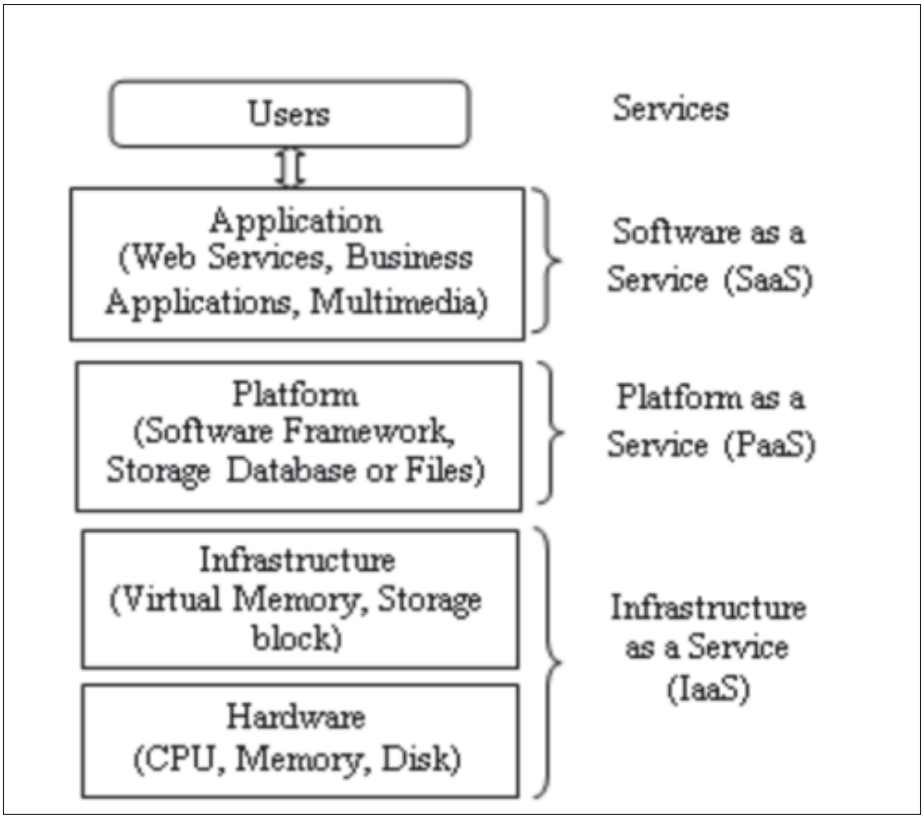

Figure 1.1 Cloud Computing Architecture taken from Kumar & Rana (2015)

- Infrastructure as a Service (IaaS) enables tenants to access computing resources within virtual machines to run the required software, for example, Windows Azure, Google Compute Engine, and Amazon EC2 (Tumkur, 2016).
- Platform as a Service (PasS), such as Microsoft Azure, and Google App Engine provides tenants with access to computing resources via an application programming interface. Customers utilize it to create and execute their applications. The user does not have access to the system's resources. The platform allocates resources to the application automatically (Tumkur, 2016).
- Software as a Service (SaaS) ,such as Google Apps, and Microsoft Office 365 offers users software programmes as a subscription service. Users are not required to install, configure, or operate the software application (Tumkur, 2016).

Based on the NIST definition, clouds can be deployed in one of four different models: Private Cloud, Community Cloud, Public Cloud and Hybrid Cloud (Mell & Grance, 2011) .

1.1.1 Public Cloud

Public cloud solutions are available from popular providers, including Google, Amazon, Microsoft, and Alibaba (Kulkarni, Aldi, Mulla, Narayan & Hiremath, 2022). Public cloud services provide the public with infrastructure and services, and you, or your organization, make a piece of that infrastructure and network secure. Resources are shared by hundreds or thousands of people (Mell & Grance, 2011).

1.1.2 Private Cloud

Private cloud solutions are dedicated to one organization or business and often have more specific security controls than a public cloud. For implementing a private cloud, there are various proprietary or open-source platforms like Apache CloudStack, OpenStack, ManageIQ, and Cloudify (Mell & Grance, 2011).

1.1.3 Community Cloud

This type of Cloud provides an infrastructure for use by a community of consumers sharing between several organizations with common concerns, including "mission, security requirements, policy, and compliance considerations." It is managed by one or a group of organizations in the community with shared interests (Mell & Grance, 2011).

1.1.4 Hybrid Cloud

This is a convergence of two or more different cloud infrastructures, including private, community or public. "they are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds)" (Mell & Grance, 2011).

1.2 Load Balancing in Cloud

Due to the high demand for online services in cloud computing, user traffic is increasing. Therefore, load balancing is required to assign tasks to virtual machines with high user satisfaction by a fair allocation of computing resources (Kulkarni *et al.*, 2022). One of the issues of cloud computing that has not been entirely addressed is load balancing. The load is like CPU, memory, and network workload. The job of load balancing is to spread the loads between different nodes of the cloud system to boost resource utilization [CPU, RAM, etc.] as well as response time considering the status of nodes; some have little work or are busy to load (Rimal, Choi & Lumb, 2009).

As can be seen in Figure 1.2, the load balancing mechanism in Cloud distributes the workload among all the nodes. The objective is user satisfaction and the ratio of resource utilization [CPU, Storage, RAM, etc.], avoiding heavy workloads on a node and improving the whole performance of the system. Load balancing in a suitable way can lead to efficient use of the available resources, scalability, and response time reduction, and maximum throughput (Swarnkar *et al.*, 2013).

Load balancer can be implemented in software or hardware format. As hardware, it acts as a reverse proxy to distribute the loads among servers (Tumkur, 2016). However, the configuration is difficult, and the device is expensive. Therefore, as software in cloud computing, which is part of SDN, Software Defined Networking, is a solution to distribute the resources dynamically (Al-Mashhadi *et al.*, 2020).

Software Defined Networking (SDN) technology is a kind of networking that employs software-based controllers or Application Programming Interfaces (APIs) to interact with hardware devices and transfer data traffic over a network. Compared with traditional networking with hardware devices such as switches and routers, SDN can build and manage a virtual or traditional network through software. However, network virtualization enables companies or organizations to have a variety of virtual networks on a physical network or vice versa, connecting hardware devices on physical networks to build a virtual network; SDN proposes a new architecture to control and route the packets of data by a central server as a controller (VMware, 2023).

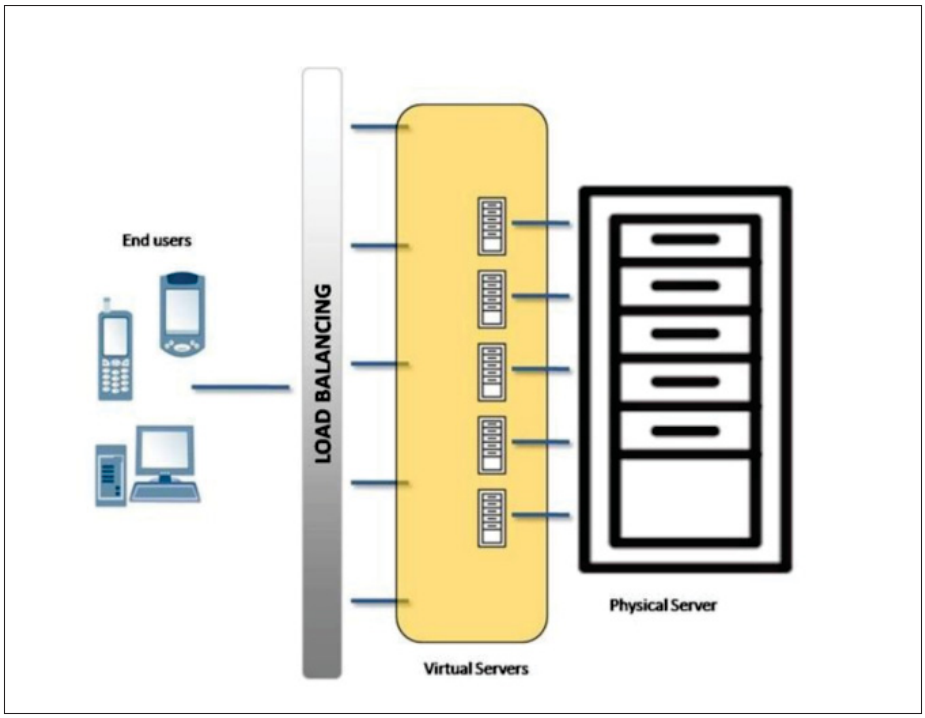

Figure 1.2 Load Balancing in Cloud Computing taken from Swarnkar *et al.* (2013)

The primary idea of SDN technology is the separation between two sections, the control plane, a programmable part [by SDN controller] and the data plane, known as the data forwarding part [by switches]. SDN reduced the traditional networking challenges in terms of management and complexity. SDN is the hottest topic among researchers and hardware producers like Ericson, Net-gear, Nokia, Siemens and Verizon in the Open Networking Foundation (ONF) list. SDN gives us a control mechanism of different network hardware devices by various vendors as a main goal. It overcomes the main issues regarding security and reliability. In the following, Figure 1.3 illustrates the SDN architecture (Al-Mashhadi *et al.*, 2020).

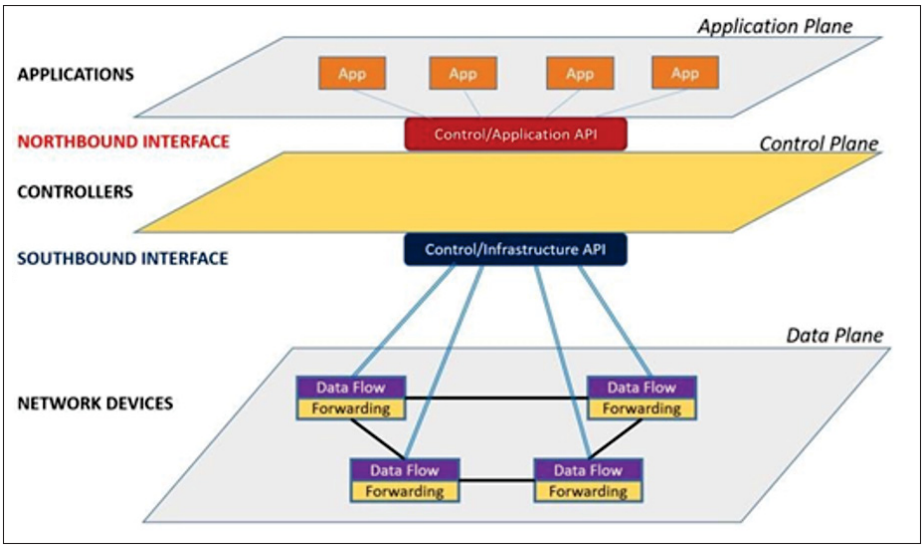

Figure 1.3 Software Defined Networking- A high-level architecture taken from Al-Mashhadi *et al.* (2020)

1.3 Load Balancing Techniques

There are two types of load-balancing techniques, Content-Aware and Content Blind, which are explained as follows: One of the techniques is the Content-Aware type. According to this technique, the data type is considered in deciding the path for transferring data packets, such as banking, gaming, website loading, etc. (Verma, 2017). It is also suitable for a homogeneous environment. In Content Blind, the data type is unimportant and compatible with a homogeneous and heterogeneous environment (Mishra *et al.*, 2020).

According to Kumar & Rana (2015), The static load balancing algorithm distributes the load equally among servers using prior knowledge of the applications and statistical data about the system. On the other hand, dynamic ones look for the lightest server in networking to assign the proper workload at run-time. Although the algorithms in this category are considered complex, they have excellent overall performance and fault tolerance.

- Content-Aware Techniques

  The Content-Aware Techniques are classified into the following methods (Verma, 2017):

1. IP Hash

   This technique makes a unique hash key through a combination of client and server IPs. This key helps to assign the request to a specific server. The key also can be reproduced if an interruption happens during the network connection (Verma, 2017).

2. DNS

   Among the various methods of DNS load balancing, DNS delegation is used to assign a domain as a sub-domain to defined zones under the control of a bunch of servers in critical situations; once servers are lost, needing to serve the requests from the website on that domain. This method works if the servers are in different areas in terms of geography over the Internet (Verma, 2017).

3. Persistence

   One of the usages of this method is in banking and security with respect to the organizations in which the HTTP connection must be connected 24/7 between the server and client. When the connection is exposed to attack or spoof in critical situations, it is essential to use this method to transfer the specific data packets, as mentioned before, by the same tunnel until the end of a session using the source and destination IP. The persistence of the session during the transferring of data packets is vital in this method which is helpful for data relating to bank or security uses (Verma, 2017).

- Content Blind Techniques

  In this technique, the data type is not as crucial as Content-Aware techniques. Compared with Content-Aware Techniques, responding to a request is much faster and is of priority. The Algorithms of Content Blind Techniques are as follows (Verma, 2017):

1. Latency

   This algorithm is common for gaming applications. In this case, the server with the fastest response is used for the required connection. It is vital that delay must not occur during the connection (Verma, 2017).

2. Round-Robin

This algorithm, known as RR, is very popular and simplest. Because the requests are assigned to servers respectively regardless of considering factors such as resource utilization, sometimes, a node might be heavy, and another stays idle without request (Verma, 2017).

3. Least Connection

   Considering the name of this algorithm, each server with a minimum number of online connections can receive the request promptly (Verma, 2017).

4. Dynamic

   This algorithm is used through an agent installed in each server to report its status. Therefore, it can be combined with other algorithms or new methods to manage an incoming request (Verma, 2017).

1.4 OpenStack

1.4.1 Cloud Computing Platforms

There are various open-source platforms for cloud computing. One of the most popular ones is OpenStack, developed by Rackspace and NASA (National Aeronautics and Space Administration), the hosting service provider, for helping cloud service providers and companies make their cloud environment. It has a modular architecture to provide tenants with different services over the Internet via a Dashboard (Huo, Qu & Wu, 2015). Many companies have used OpenStack for their business goals. Data centers also use OpenStack to provide computing, storage, and networking (Harvey, 2014).

1.4.2 OpenStack Architecture

OpenStack plays a crucial role for a cloud provider with a powerful dashboard to manage servers on different hypervisors (VMware, ESXI, KVM, Hyper-V, Xen, etc.) as well as required storage and virtual network services. It enables cloud customers to use various services, including

network, storage, and computing. The OpenStack architecture consists of four categories (Callegati, Cerroni, Contoli & Santandrea, 2014):

- **Controller nodes** to manage the cloud platform (OpenStack, 2023g). Controller nodes perform API (Application Programming Interface) for the components of OpenStack, such as Cider, Glance, Nova, Keystone, and Neutron. These API services install on multiple controller nodes if needed (Denton, 2016).
- **Network nodes** for providing cloud network services (Callegati *et al.*, 2014). Network nodes perform the network services such as DHCP and metadata and provide virtual routers if the Neutron L3, DHCP and metadata agents are installed (Denton, 2016).
- **Compute nodes** usually execute the VMs on a hypervisor, including KVM, Hyper-V, Xen, etc (Callegati *et al.*, 2014).
- **Storage nodes** are related to 3 types of storage, like Cider, Swift, Or Glance for data and VM images (Callegati *et al.*, 2014).

Figure 1.4 shows the Cloud Architecture in the following:

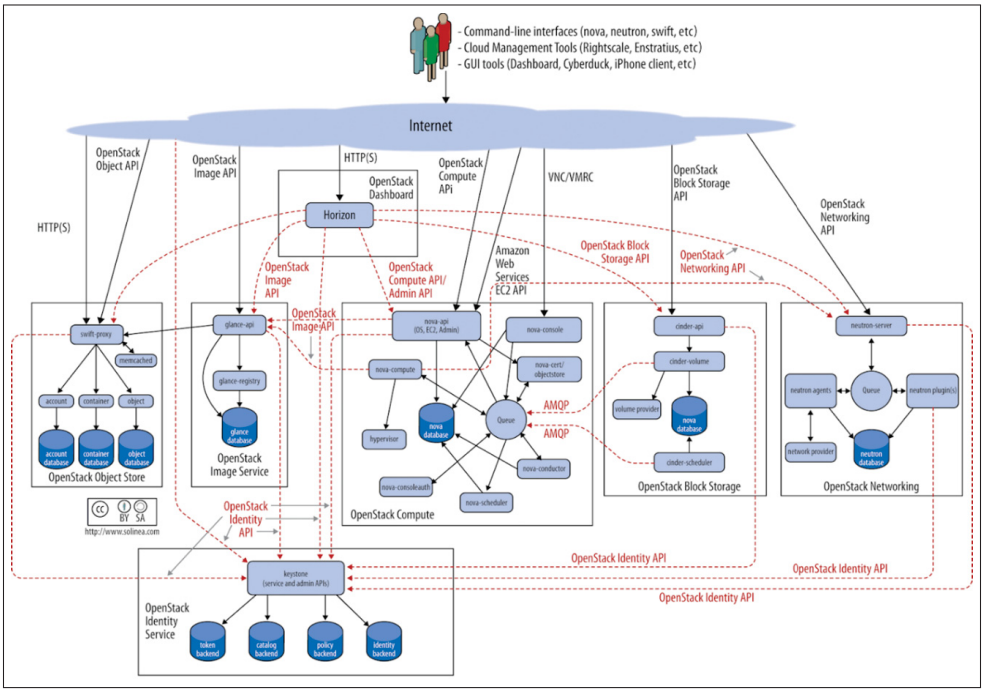

Figure 1.4 OpenStack Logical Architecture taken from OpenStack (2018b)

1.4.3 OpenStack Components

In the following, OpenStack has different components (OpenStack, 2023i):

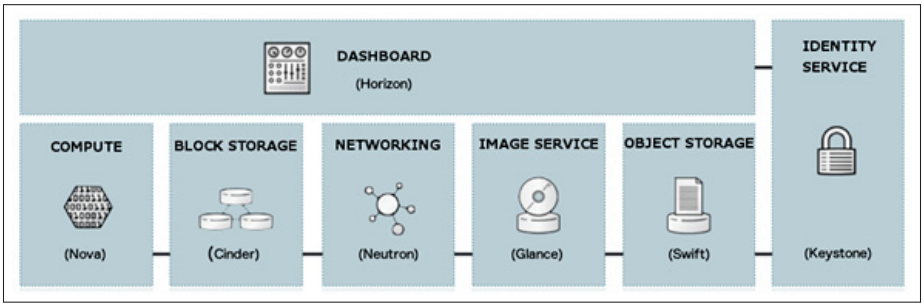

Figure 1.5 OpenStack Service Overview taken from OpenStack (2023i)

Figure 1.5 presents various components of OpenStack. As a brief explanation, Nova, as an OpenStack compute service, manages the virtual machine instances, networks and tenants on Cloud.

Neutron, as a networking service, provides different networking services such as DHCP, load balancing, firewall, etc.
Keystone, as an Identity service, is used for authentication and authorization services in which different network nodes can communicate via a token.
Other components are related to storage services, such as **Cider** (Block storage service) is used for block devices such as a hard disk.
Swift (Object storage service) is used for media, backup, and VM image files.
Glance (Image service) is used for managing image services such as discovery, registration and delivery.
Another component, **Dashboard** (Horizon), provides an interface for cloud customers (administrators and users) to manage and monitor the resources on Cloud (OpenStack, 2023i).

OpenStack services can correlate by calling the Application Programming Interface (API) of each service to interact with others in a modular manner in a loosely coupled system. Figure 1.6 depicts the correlation between OpenStack modules (Huawei Technologies Co., Ltd., 2023).

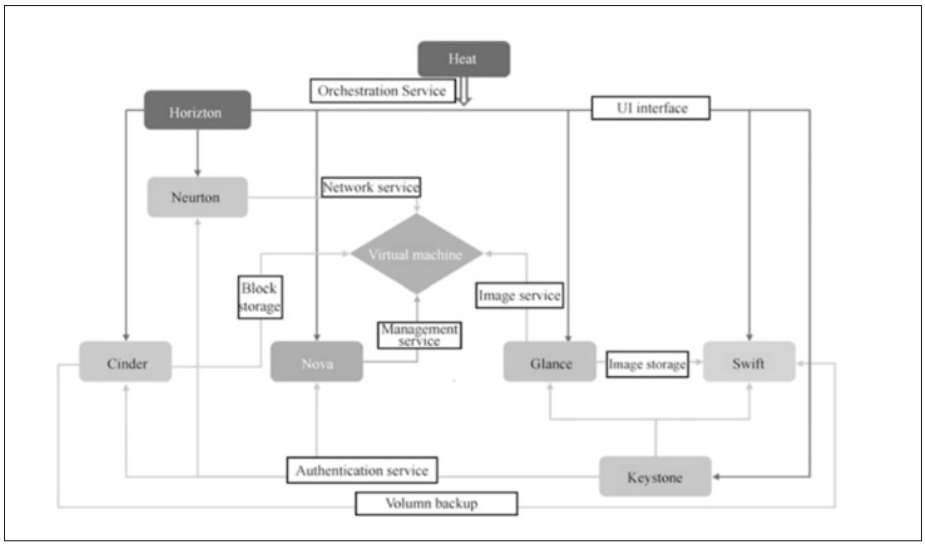

Figure 1.6 The logical interaction among OpenStack components taken from Huawei Technologies Co., Ltd. (2023)

1.4.4 Computing Service (Nova)

Nova, the control unit of OpenStack, is responsible for managing servers, creating virtual machines and supporting containers. It correlates with other components of OpenStack for different approaches, such as authentication with Identity service, three types of storages with Glance, Swift, and Cider, network services with Neutron service, etc. (OpenStack, 2021).

Figure 1.7 depicts the interaction of Nova with its components and other modules in OpenStack. Nova is crucial in managing and selecting the best hypervisors using the API server. It is possible to build clouds with multiple hypervisors in various zones. The Hypervisors such as Hyper-V, VMware vSphere, Kernel-based Virtual Machine (KVM), QEMU, Xen, Linux Containers (LXC), etc, are supported by Compute (OpenStack, 2021).

The main components of Nova-compute are as follows (OpenStack, 2021):

- **Nova-API** is in charge of receiving all the requests coming from other components to Nova and responding to them through APIs.
- **Nova-compute** is for managing instances on compute node.

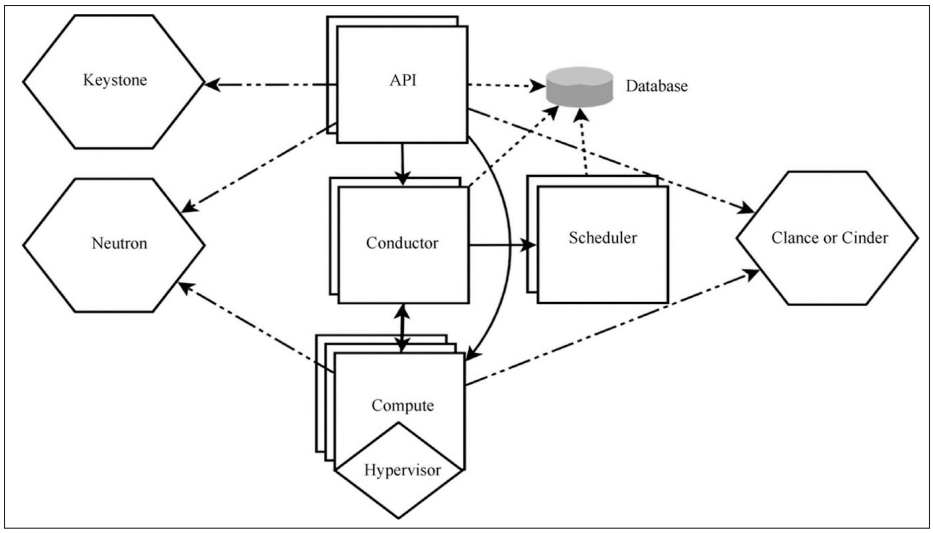

Figure 1.7 The logical interaction among OpenStack components taken from Huawei Technologies Co., Ltd. (2023)

- **Nova-scheduler** is responsible for choosing the appropriate compute node for hosting the VM.
- **Nova-conductor** frequently requires database updates, like updating and retrieving the virtual machines' status. Nova-compute does not directly access the database for security and scalability reasons, delegating this operation to Nova-conductor. It has two advantages: increased system security and improved system scalability (Huawei Technologies Co., Ltd., 2023).

According to Figure 1.8, when Nova API receives a request to build an instance, some essential process is performed for sending a message to Messaging by RabbitMQ. By this message, Scheduler starts scheduling to choose one of the most suitable compute nodes. Then Scheduler sends a message to messaging to create an instance on the specified compute node. Through this message, Nova-compute starts the instance on the compute node. Meanwhile, a message is sent to Nova-conductor if it needs database access for updates or queries since the conductor can communicate with the database.

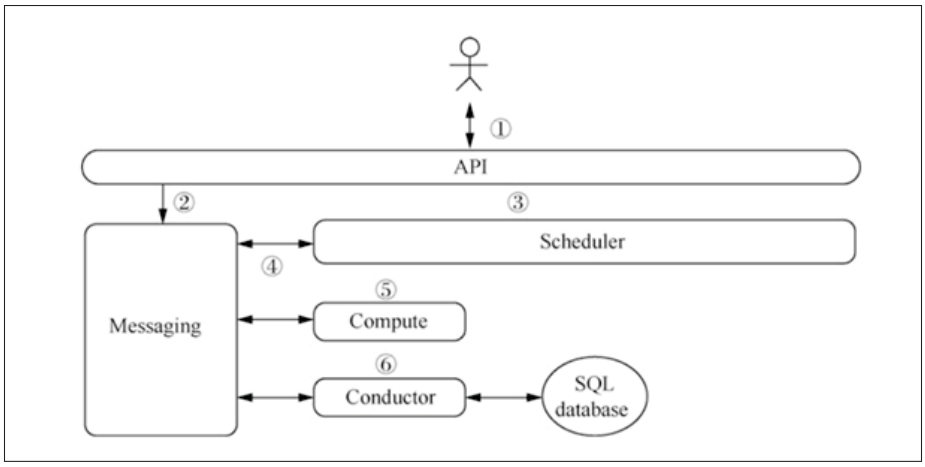

Figure 1.8 Nova Workflow taken from Huawei Technologies Co., Ltd. (2023)

1.5 Networking Service (Neutron)

OpenStack Networking, Neutron, acts as an API to manage virtual and physical networks over OpenStack Cloud. The main purpose of Neutron is connecting the virtual machines to the virtual network on the Cloud and then connecting the virtual network to the physical network. Network physical devices such as switches, routers, firewalls, and load balancers can be configured virtually (Denton, 2016)

Neutron enables the tenants to employ and use networking resources. In (OpenStack, 2023e) , it was stated that "OpenStack Networking provides a tenant-facing API for defining network connectivity and IP addressing for instances in the cloud, in addition to orchestrating the network configuration."

According to Figure 1.9, Neutron architecture includes Neutron-API, plugins, and agents that are described below:

Neutron API on the Neutron server receives demands and sends them to plugins for network implementation.

Core plugins are used for supporting the two-tier virtual network by the L2 agent via OVS (Open vSwitch). Since there is much more code duplication in implementing core plugins, the ML2 core plugin is used instead of core plugins (Huawei Technologies Co., Ltd., 2023). The

ML2 plugin depends on various drivers' types to define network types such as flat, VLAN, VXLAN, and local, supported by Neutron Denton (2016).

Service plugins provide network services such as routing, load balancing, and firewalling (Huawei Technologies Co., Ltd., 2023).

Neutron Agents such as DHCP, MetaData, L2, and L3 are used on compute and network nodes to implement a network by receiving the messages on the message bus from the Neutron server. DHCP is for providing DHCP services to all networks, and MetaData services provide users with instances' information such as hostname, Ips and public SSH keys (Denton, 2016).

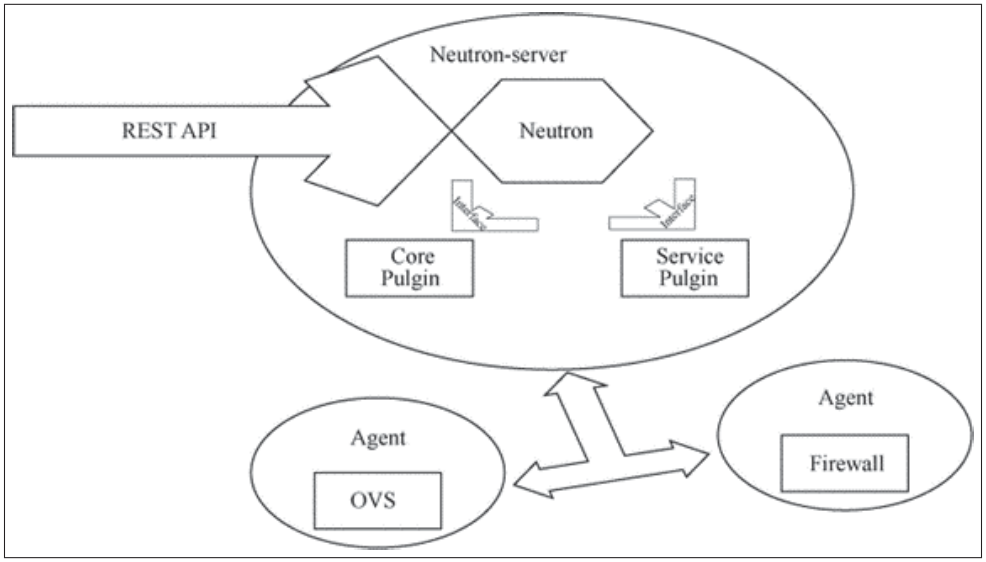

Figure 1.9 Neutron architecture taken from Huawei Technologies Co., Ltd. (2023)

Figure 1.10 shows the correlation between agents and the Neutron server to create a virtual network.

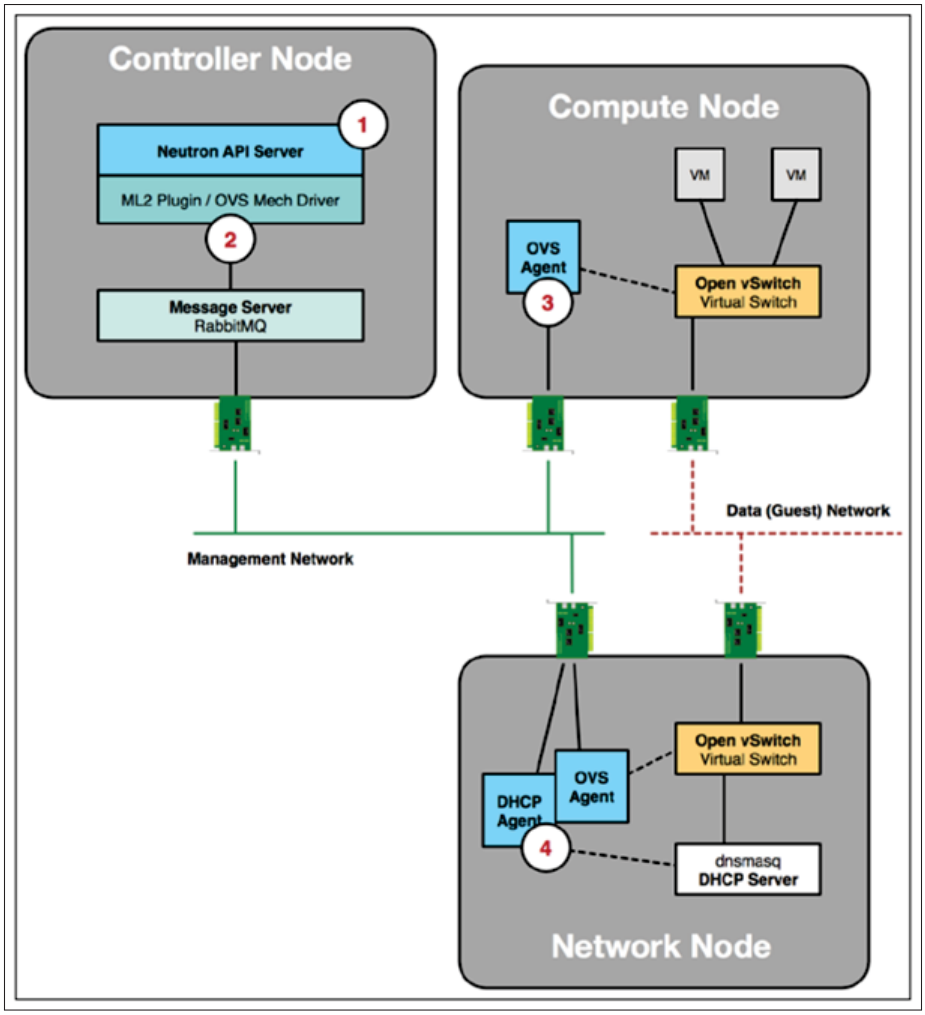

Figure 1.10 The interaction between network plugin agents and the Neutron server taken from Huawei Technologies Co., Ltd. (2023)

Based on Figure 1.10 , at the first step, a request is received by Neutron API for connecting an instance to a new network. Then To process the request, the API server calls the ML2 plugin. In the second step, The ML2 plugin forwards the request to the OVS mechanism driver, which generates a message based on the information in the request. The message is forwarded to the appropriate OVS agent for processing via the management network. Next, The message is received by the OVS agent, who configures the local virtual switch. In the last step, receiving the request is received by the DHCP agent to configure the DHCP server within the network

node. Then virtual machine instances will be assigned IP addresses over the data network by the DHCP server (Denton, 2016).

As mentioned above, in Neutron, there are three services such as LBaaS (Load Balancing as a Service, FWaaS (Firewall as a Service) and VPNaaS (Virtual Private Network as a Service) (OpenStack, 2023f). In this report, LBaaS is considered.

1.6 Load Balancing in OpenStack

OpenStack Neutron can distribute incoming requests between configured instances. Using Neutron and OVS (OpenvSwitch), Load Balancing-as-a-Service (LBaaS) can be built. The load balancing of workloads is used to spread incoming traffic between designated instances. This operation ensures that no server takes all requests leading to a slowdown. Therefore, traffic is divided among active instances and uses resources effectively. OpenStack LBaaS includes the following methods (OpenStack, 2018a).

- **Round robin** has a simple mechanism that executes processes in circular order between different instances regardless of priority.
- **The source IP** method assigns client requests via specific IPs to pass to the same instance.
- **The least connections** method is used for sending incoming requests to the server with fewer active connections.

1.7 Load Balancer as a Service (LBaaS V1, V2)

Initially, LBaaS V1 was presented as a feature by the networking service in OpenStack; then, it was updated to LBaaS V2 with a plugin named "neutron-lbaas". LBaaS V2 has a facility named listener that lets you set multiple listener ports, such as 80 and 443, on a load balancer IP address based on Figure 1.11 (OpenStack, 2023h).

LBaaS V2 can be implemented in two ways, API-based via Octavia and agent-based via an external 3rd party component like HAProxy software server (High Availability Proxy) (OpenStack, 2023h).

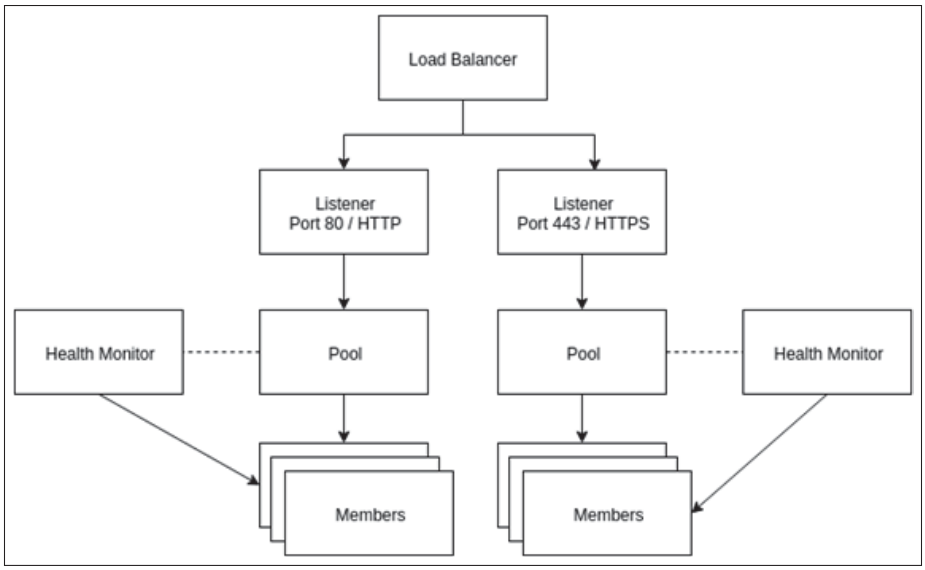

Figure 1.11 LBaaS V2 concepts taken from OpenStack (2023h)

HAproxy software is limited for TCP and HTTP-based applications ("e.g. high traffic websites") as an open-source high availability load balancer and proxy server proxying solution roles to distribute requests between servers in cloud platforms (HAProxy, 2023).

Octavia is a superset of LBaas V2 API, including additional features. It is used as an open-source load balancer using a separate API inside virtual machines on hypervisors which is manipulated by the compute service. In this case, there is no need for an agent (OpenStack, 2023h). According to the mentioned load balancing methods in section 1.4.6, Octavia supports three Load balancing methods: Round robin, Source IP and least connection methods.

In 2019, neutron-lbaas and neutron-lbaas-dashboard are retired and no longer supported as of the Queens OpenStack release cycle. There are many reasons behind that. One and foremost is that neutron stadium has grown considerably, resulting in some management scaling issues. By this, neutron-lbass was considered as a high-level OpenStack project. Second, there is no need to access the neutron code because of APIs. Through this change, Octavia uses independent APIs to interact with other services of OpenStack. Moreover, the performance of load balancing API was increased by removing neutron API code. Also, the number of repositories for adding API features is decreasing to two repositories. Another is that; some substantial flaws in the

neutron-lbaas API code lead to the inconsistent state via a high volume of API calls which resulted in renewing API code provided by Octavia. Lastly, Octavia, as an independent project, can update quickly, and no one is confused by the name "neutron" (OpenStack, 2023g).

1.8 Octavia Project

After the Neutron-LBaaS project, which was updated from version 1 to version 2, Octavia is replaced for implementing LBaas V2 API comprising additional features against neutron-lbaas after the Liberty release of OpenStack. Now, Octavia, as an official OpenStack service project provides load balancing capabilities for OpenStack with a standalone API which is considered in Keystone as load balancer service (OpenStack, 2023d).

Octavia can provide load balancing services by managing VMs, containers, or bare metal servers called amphorae setting up on-demand. With an on-demand, horizontal scalability feature, Octavia is more compatible with the cloud environment than the other Load balancing solution due to high availability by amphorae. In Octavia, Amphora is a virtual machine instance located in Compute node including HAproxy software to act as a load balancer in the Octavia module. And amphorae is more than one Amphora. Therefore, using the other modules, Octavia plays a crucial role in the OpenStack ecosystem. In detail, Octavia interacts with the following modules and libraries for its role. More explanation will be proposed below: (OpenStack, 2023d)

Table 1.1 shows the modules and libraries collaborating with Octavia to provide tenants with load balancing services.

Table 1.1 Interaction of Octavia with OpenStack Modules

Modules and Libraries	Controller	Compute	Network
Keystone Module	✓		
Glance Module	✓		
Nova Module	✓		
Neutron Module	✓	✓	✓
Barbican Module	✓		
Oslo Messaging Library	✓		
TaskFlow Library	✓		

- **Nova** for controlling Amphora and managing the compute resources on request.
- **Neutron** for network connection among Amphora, tenant environment and external network.
- **Barbican** for manipulating TLS[6] (Transport Layer Security) certificates including private keys to secure connections between clients and backend nodes.
- **Keystone** for authentication to the Octavia API as well as Octavia with other moudules.
- **Glance** for maintaining the amphora instance image.
- **Oslo and taskflow** as part of Oslo for making communication between Octavia components.

1.9 Octavia Components

According to Figure 1.12, Octavia consists of some components and subcomponents. There are three components. First, amphorae are the VMs, containers, or bare metal servers to provide tenants with load-balancing services. Second, the Controller, as Octavia's "brains, " comprises five sub-components: API Controller, Controller Worker, Health Manager, Housekeeping Manager, and Driver Agent. Last is Network, which is needed for Amphora to spin up to load balancer network and reach the tenant network's backend members via a network interface (OpenStack, 2023d).

The subcomponents of Octavia are as follows:

- **API Controller** is responsible to take requests, perform and transfer them to the controller worker over Oslo messaging bus.
- **Controller Worker** is in charge of receiving the requests from API controller and taking necessary action to fulfill the request.
- **Health Manager** is for monitoring amphorae to make sure they are up and running. Also, it manages failover events if one of the amphorae fails.
- **Housekeeping Manager** is for cleaning the deleted database records and managing the spare pool.
- **Driver Agent** is for connection with other modules.

---

[6] TLS is used to provide secure communication over network.

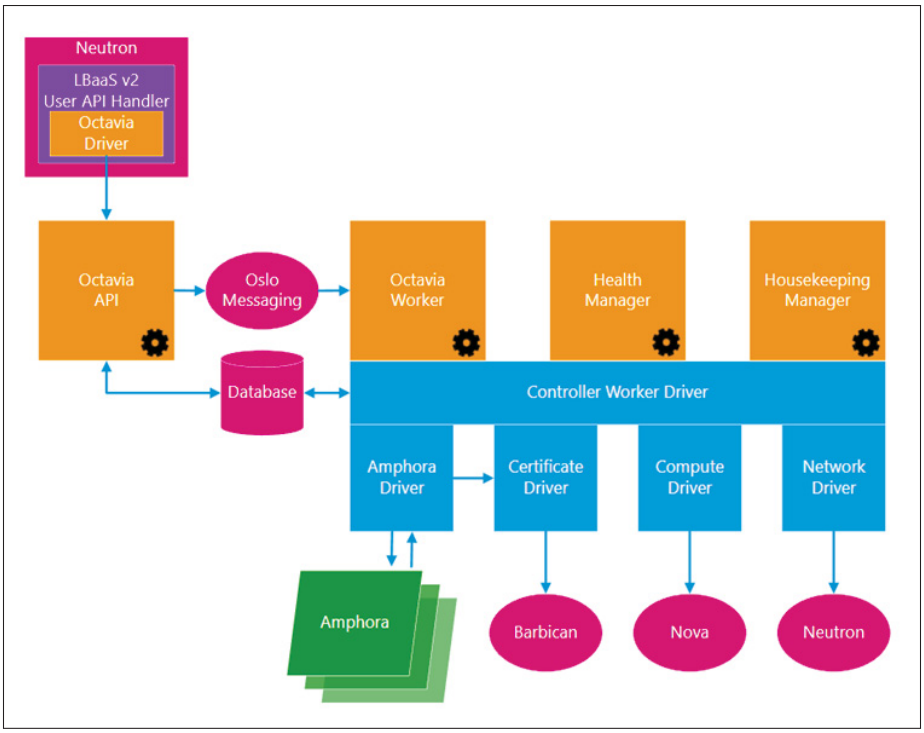

Figure 1.12 Octavia components taken from OpenStack (2023d)

CHAPTER 2

LITERATURE REVIEW

We are going to review the works related to our study. In a distributed environment, load balancing is needed for better performance and efficiency and is vital for Cloud Computing in the research area. There are different heuristic load-balancing techniques. We will concentrate on papers involving dynamic methods in the cloud environment through different simulations. The target is to survey the various contributions with their cons and pros for adding a dynamic version to the OpenStack platform.

2.1 Dynamic Load Balancing Algorithms in Cloud Computing

Assigning the user requests to the specific servers on cloud networks can lead to some challenges, including power usage or run time from overloading or underloading. In this case, to overcome the mentioned challenges, load balancing is a mechanism to balance different types of loads in the Cloud, including CPU, RAM and network load. There are various algorithms for load balancing consisting of dynamic for "homogeneous and heterogeneous environments" and static for "homogenous or stable environments" (Tsai & Rodrigues, 2014). Based on the mentioned problems, different algorithms are reviewed and discussed. Before going through the existing algorithms, some performance metrics show the performance of various load-balancing algorithms. Among these performance parameters in the Cloud, Makespan and energy consumption are important. Makespan definition is the whole time needed to complete all tasks submitted to the system (Mishra *et al.*, 2020). Energy consumption is the energy consumed by all ICT devices connecting in the cloud system (Moganarangan, Babukarthik, Bhuvaneswari, Basha & Dhavachelvan, 2016)

Regarding the existing load balancing algorithms and their issues in Cloud Computing, five parameters are considered: Scalability, Fault Tolerance, Throughput [7], Resource Utilization,

[7] Throughput shows the number of executed tasks per unit of time. It should be high in order to optimise system performance (Swarnkar *et al.*, 2013).

Point of Failure and Response Time (Swarnkar *et al.*, 2013). Based on challenges such as avoiding overloading, latency, high energy consumption and response time coming from load balancing, much research has been done to address the problems.

In 2010, a dynamic load-balancing algorithm named Honeybee Foraging Behavior (the behaviour of bees inspired this name to seek their food) was investigated by Randles et al. In this method, although system performance is raised via increasing system diversity, the throughput parameter of this algorithm is not improved. It is suitable for situations requiring a diverse population of service types. In this algorithm, virtual servers serve the requests and calculate their profits based on CPU utilization to compare with other servers. The server with a high profit is selected to suit the demands; otherwise, another server chooses randomly (Randles, Lamb & Taleb-Bendiab, 2010). This algorithm was proposed by (Nakrani & Tovey, 2004).

In 2013, Dasgupta et al. offered a load balancing method using Genetic algorithm with "the mechanism of natural selection strategy" for cloud computing to make resource utilization efficient. This method balances the workload based on a minimum makespan of coming tasks using the CloudSim simulator. Although the proposed strategy performed better than existing techniques (FCFS [8] , and RR [9]), all the requests have the same priority, which may not occur in a real case. Therefore, this algorithm is inefficient due to a huge number of VMs and demands. Moreover, As this algorithm has a simple approach of GA, selection strategies can be changed to make it more efficient (Dasgupta, Mandal, Dutta, Mandal & Dam, 2013).

In 2013, Wu et al. studied on WRR (Weighted Round Robin) and WLC (Weighted least connection) to build AWLLB, an Adaptive Weighted Least Load Balancing algorithm. In this method, once the load balancer received the request, server weight could be dynamically adjusted and selected the nodes with the minimum amount of weight based on CPU usage to serve the request. Then, the authors compared the performance of AWLLB with WRR and WLC through the CPU utilization ratio in the OPENT platform. According to this benchmark, in AWLLB,

[8] (First Come First Serve)

[9] (Round Robin)

the CPU utilization ratio is lower than WRR and WLC. In WRR, high CPU usage resulted in bottlenecks due to congestion on one server while other servers were vacant (Wu, Luo & Li, 2013).

In 2015, Singh et al., investigated the issues in terms of scalability and reliability and static algorithm in the mentioned artificial intelligent algorithms and proposed "an Autonomous Agent-based Load Balancing algorithm (A2LB) which can offer maximum resource utilization, maximum throughput, minimum response time, dynamic resource scheduling with scalability and reliability." This mechanism consists of agents calculating the proactive load of each VM in a data center. If a VM's load approaches a threshold amount, the Channel and Migration agents search for a potential VM from other data centers. These agents behave like ants seeking the shortest and best route to the destination. As the ants try to look for different paths to the destination, they record the routes with a chemical material, then after a while, this chemical material increases. Therefore, other ants follow that as the best and shortest path to the destination. While moving from source to destination, they collect the path information and do not return to their source, reducing traffic. Consequently, these agents are suitable for load balancing in Cloud Computing with different data centers to seek the underloaded servers (Singh *et al.*, 2015).

In 2017, Li et al. designed and implemented a load balancing technique on the OpenStack platform based on load forecasting model and BP neural network algorithms. The algorithm includes three steps, monitoring and gaining load information, load forecasting model and virtual machine scheduling. Therefore, by this method based on BP neural network, load fluctuation is slight, and network accuracy by online learning is approved (Li, Zheng, Li, Xu & Tang, 2017).

In 2018, Xinming et al. studied the merits and demerits of static and dynamic load balancing algorithms in distributing services to propose a dynamic load balancing algorithm. In this paper, this algorithm employs the real-time load on each server by combining the server's performance and the request number. Then it updates the weights through the dynamic feedback mechanism, which minimizes average response time and improves the server's overall utilization rate (Rong He, Xinming Tan, 2018).

2.2 Summary

Based on the literature review, it has been observed that there are different dynamic mechanisms in load balancing for Cloud environments with cons and pros in terms of scalability, reliability, and throughput. To reach our goal, we plan to add a dynamic solution of load balancing algorithm based on the static method, Weighted Round Robin via OpenStack.

CHAPTER 3

DESIGN AND IMPLEMENTATIONS

To reach the objective of adding an agent-based load balancing in OpenStack, we prepare a cloud environment by installing OpenStack to built-in an agent-based algorithm over a standard method on the load balancer. The steps to implement the agent-based method will be described.

3.1 Installation Environment

For implementing the proposed solution, the experiment is performed in a private cloud environment via OpenStack on VirtualBox hypervisor to virtualize the environment. It is installed on hardware as a host, including AMD Ryzen 5700U Processor and 16GB RAM. To make a cloud environment via OpenStack, three virtual machines, Controller, Compute and Network, are installed with Ubuntu Linux 20.04 with different configurations according to Table 3.1.

Table 3.1 Hardware Configuration

Virtual Device	Controller	Compute	Network
CPU (Cores)	2	2	2
RAM (GB)	4	6	4
Primary Disk (GB)	20	40	20

For our purpose, the OpenStack architecture consists of three nodes with its tasks. The Controller node manages the cloud platform, managing and maintaining APIs for the modules on OpenStack. The Compute node is for installing, executing, and managing the virtual machine instances as web servers, and the Network node provides instances with network services. Moreover, for better performance, the network traffic is segregated into Control Plane for managing traffic and Data Plane for data user traffic.

Figure 3.1 illustrates our Cloud environment in OpenStack platform.

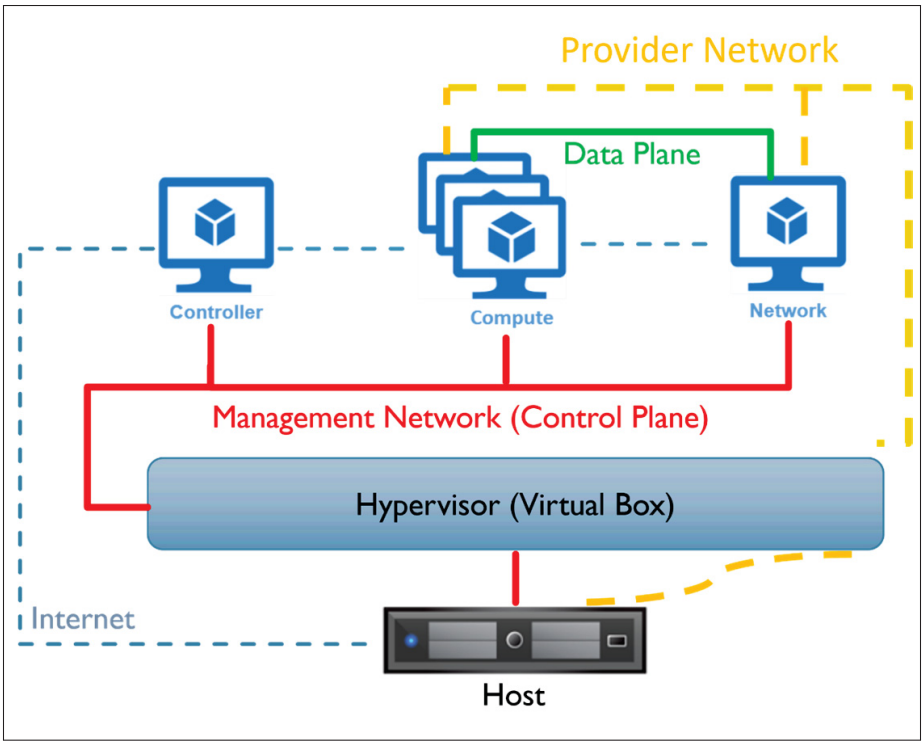

Figure 3.1 Installation Environment (Cloud Environment)

Basic OpenStack packages like Yoga repository and python-OpenStackclient 5.8.0 Version in all nodes are installed to prepare the infrastructure. Then, some requirements such as MariaDB for managing SQL Database, RabbitMQ for controlling Message Queue and Memcached for cache tokens are installed.

Based on Figure 3.2, for running the OpenStack platform, essential modules such as Keystone for identity services, Glance for image services, Nova (API, Conductor, and Scheduler) as the control unit of OpenStack, and Neutron for network services are installed on the Controller node. The Neutron plugin agents (OpenVSwitch, DHCP, and L3) are installed on Compute and Network nodes to build L2 and L3 connectivity between private and provider networks to reach the external network to serve the users' requests. Afterward, we must create a nested virtualization network by Nova compute, QEMU emulator and OpenVSwitch as a virtual switch to configure the network among virtual machine instances on Compute node. Moreover, each

node has a separate interface to connect to the Internet to set up the cloud environment and update the repository. More information is placed in Appendix I.

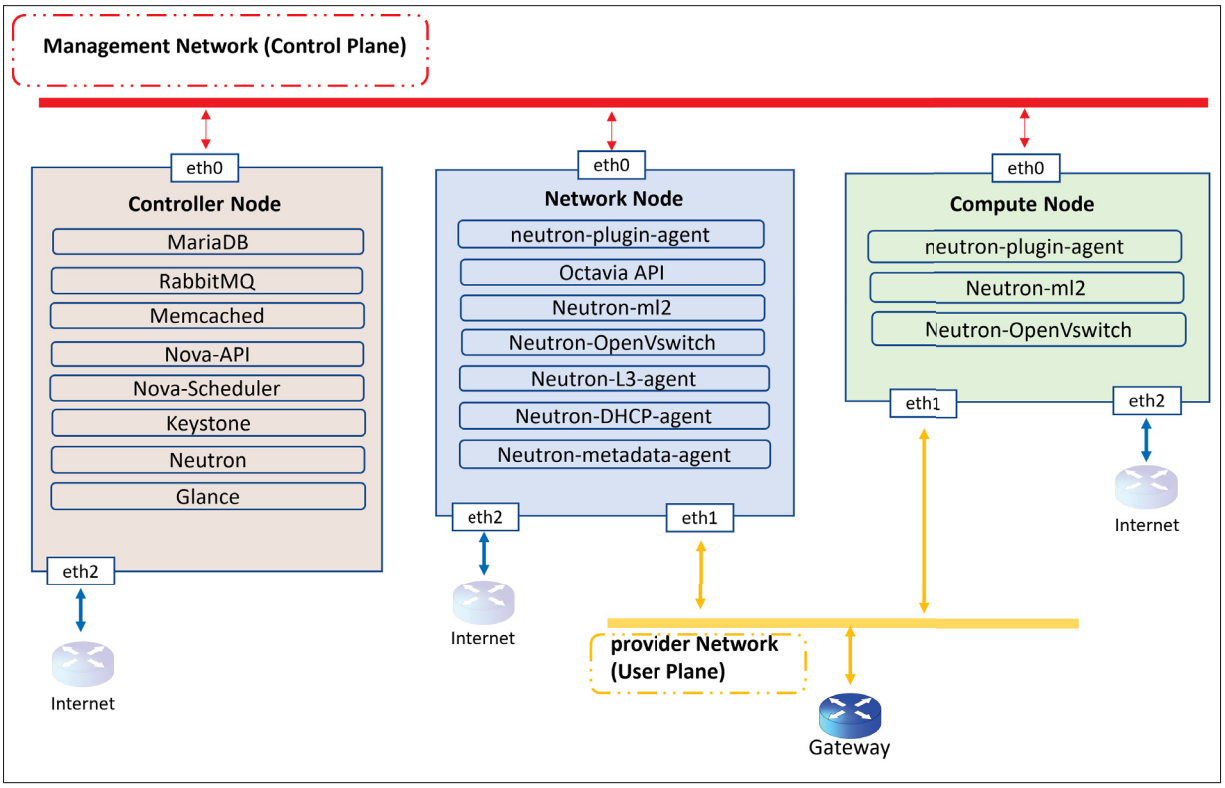

Figure 3.2 Network Service Architecture

Once all essential modules, virtual switch and QEMU emulator are installed, we create a network topology including three instances within Compute node with private network.

In this thesis, three web servers, WS1 (WebServer1), WS2 (WebServer2), and WS3 (WebServer3), based on Table 3.2, are installed and configured as web servers by installing Apache2.

Table 3.2 Backends' Hardware Configuration Details

Device	WS1	WS2	WS3
CPU (Core)	1	1	2
RAM (GB)	1.00 GB	2.00 GB	2.00 GB
OS [10]	Ubuntu 20.04	Ubuntu 20.04	Ubuntu 20.04

[10] Operating System

3.2 Deployment

After making the cloud environment via OpenStack, we deploy the Octavia module as necessary as other modules such as Keystone, Nova, Neutron and Glance for Load balancing services in the Network node. As mentioned earlier, Octavia API V2 by Neutron_LBaaS was deprecated and replaced with the Octavia project.

Octavia service and the following components are created and configured in this infrastructure based on Table 3.3.

Table 3.3 Octavia Components

Components	**Node**
Amphora	Compute
API Controller	Network
Octavia Driver	
Controller Worker	
Health Manager	
Housekeeping Manager	

Octavia plays a load balancer role between the web servers and clients' requests based on the WRR (Weighted Round Robin) algorithm. The web servers can connect to the load balancer by joining the pool, and clients consider it the web sites' server address to get the requests' answers.

In Figure 3.3, for load balancing as a service via Octavia, all management traffic use the management network interface through messaging queue to connect the related agents and APIs to run the load balancer within Amphora VM in compute node. Inside the amphora, there is HAproxy software as load balancer, which is used for load balancing to set listener, pool member and algorithm. Amphora is managed and monitored by Octavia API components through the managemnt network. Amphora VM as a load balancer and web servers' virtual machines communicate togheder through OpenVswitch and then connect to the provider network to reach the external network by floating IP on the virtual north-south router.

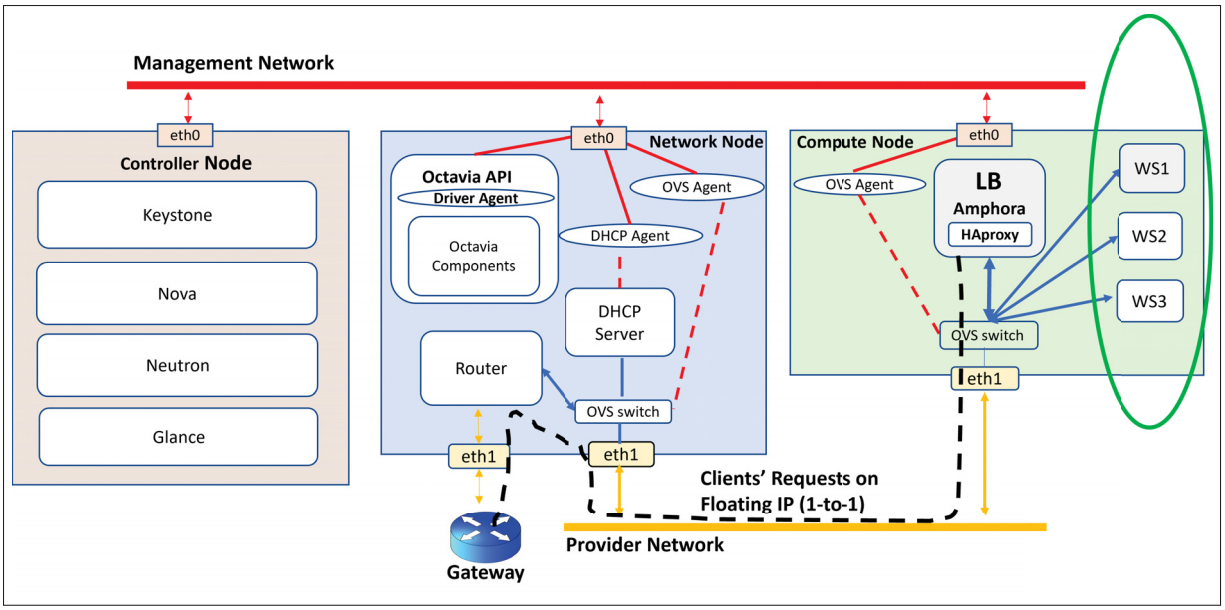

Figure 3.3 Load Balancer as a Service (LBaaS) via Octavia

According to the servers' configuration, different weights are assigned to web servers described in Table 3.4. We set the maximum weight to WS2 and the minimum weight to WS1. This configuration shows the difference between static and dynamic load balancing in the next chapter.

Table 3.4 Servers' weights

Host Name	Numeric Weight
WS1	2
WS2	10
WS3	3

3.3 Simulation of Workload via Multi Threads

For sending requests to the load balancer, we execute the Python script, which simulates the clients' HTTP requests (via socket library) in different loops via multi threads. In the first step,

this script gets the number of requests and threads as input values to calculate the load per thread based on the following equations[11]:

Suppose the number of requests is R, and the number of threads is T.

- T = the number of threads
- R = the number of requests

In Equation 3.1, L_i is calculated the load of the No .i thread.

$$L_i = \lfloor R/T \rfloor \tag{3.1}$$

Equation 3.2 presents the S_L, which is the whole burden of all threads except the last thread.

$$S_L = \Sigma_1^{T-1} L_i \tag{3.2}$$

Equation 3.3 expresses the L_T as a load of last thread.

$$L_T = R - S_L \tag{3.3}$$

At the Second step, the initial time is calculated. Then we create a thread pool[12] with an input number of threads to manage the parallel threads for distributing the sequence of loads among them. Next, we assign a function called Requester as well as the sequence of loads in form of a list in parallel to the thread pool. By this, the Requester function sends each element of load sequence per thread to load balancer in parallel through HTTP GET method on TCP[13] socket. At the final step, the ending time is estimated.

---

[11] These equations are helpful when the Quotient of the number of requests divided by the number of threads will not be integer. Therefore, by this way, we can process all requests via multi threads to have a better simulation of workload coming from clients.

[12] Thread pool is used for multiprocessing.

[13] TCP (Transmission Control Protocol)

In the following, Figure 3.4 shows an example of load calculation via multi threads.

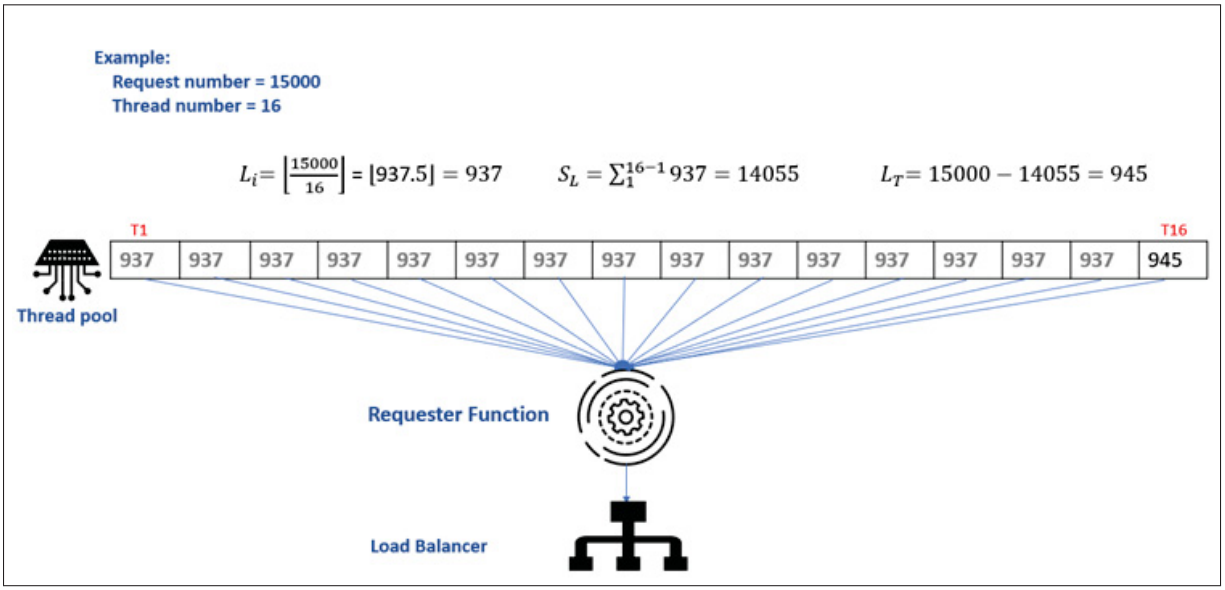

Figure 3.4 Load Simulation via Multi Thread

3.4 Weighted Round Robin Load Balancing (Existing Static Method)

After the installation phase, based on Figure 3.5, we deployed Octavia as a load balancer over one of the popular static methods, WRR. There are some reasons behind that. One is that the allocation of requests is simple and easy to use. Second, server's real-time status is not considered by this algorithm, and the weight does not accurately reflect the server's performance.

Weighted Round Robin builds on simple Round Robin load balancing. In the weighted version, a static numerical weight is assigned to each server based on factors like CPU usage. Initial weights, 2,10 and 3 are assigned to WS1, WS2, and WS3, respectively.

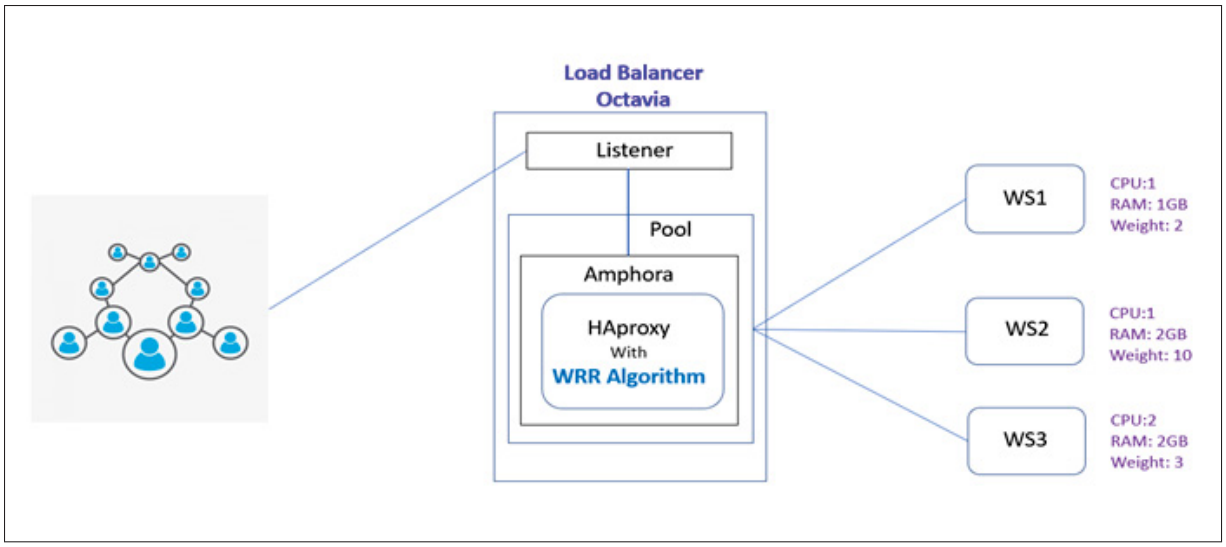

Figure 3.5 Load Balancer with Static method, WRR

Based on the logs below in Figure 3.6, it is visible that the WRR algorithm distributes the requests blindly regardless of the last status of each servers' CPU usage due to the large gap among CPU percentages of WS1, WS2, and WS3. Large differences between servers' weights leading to requests congestion on one server. For example, in the logs, WS2, with 100% CPU usage, is busier than WS3, with 21%.

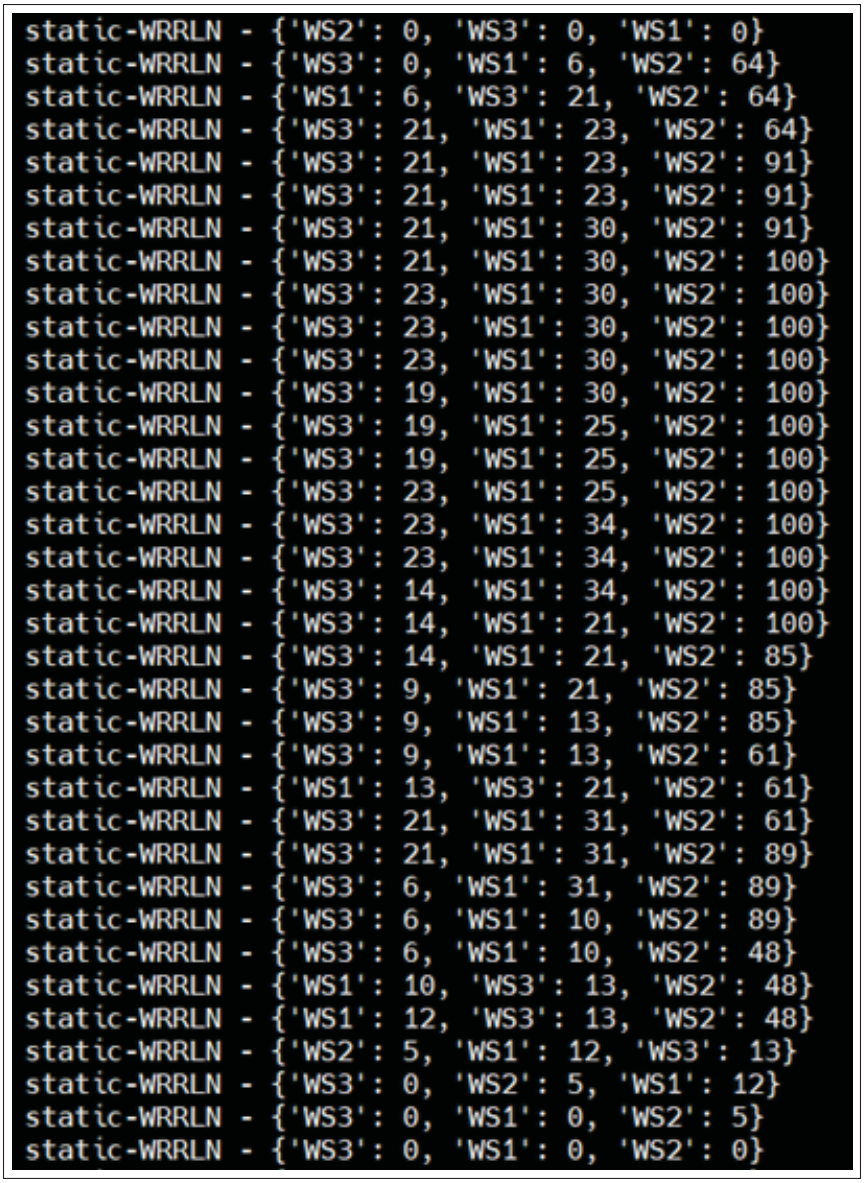

Figure 3.6 Logs for executing WRR Load Balancing

Some limitations of this algorithm are as follows:

- By the static method, the server's performance can be poor if the server is busy with local tasks such as any updates on an operating system, OS functionality or hardware issues such as CPU, memory and cooling.

- Network administrators must manually change weights on servers. Human variables significantly impact weight setting, and static weight values cannot accurately represent the servers' dynamic demand and real-time processing capabilities.
- In the case of the same weights with different configurations, the servers are not being used to their full potential, resulting in resource waste.
- If there is much more difference between the servers' weights, then request congestion will happen in one of the servers.
- This algorithm does not consider the real-time status of each server.

3.5 Dynamic (Agent-based) Load Balancing (Modified Static Method)

This algorithm differs from the WRR which was mentioned above. Despite the static method, in the agent-based method, the load balancer always considers the last status of each server at run time. In contrast, in the static mode, the initial state is considered to choose the lightest server.

To overcome the mentioned limitations of WRR, an agent is used on web servers to send the last servers' status to the load balancer for updating the weights dynamically. We use the same network topology with three web servers utilized in the previous method but with an additional agent.

For simulation purpose, based on Figure 3.7, we have implemented an agent located on each web servers to collect servers' statistics by Python script to send the run-time information such as hostname, host IP and CPU usage percentage to the load balancer every 1 second on UDP[14] protocol due to prevent latency.

[14] UDP (User Datagram Protocol)

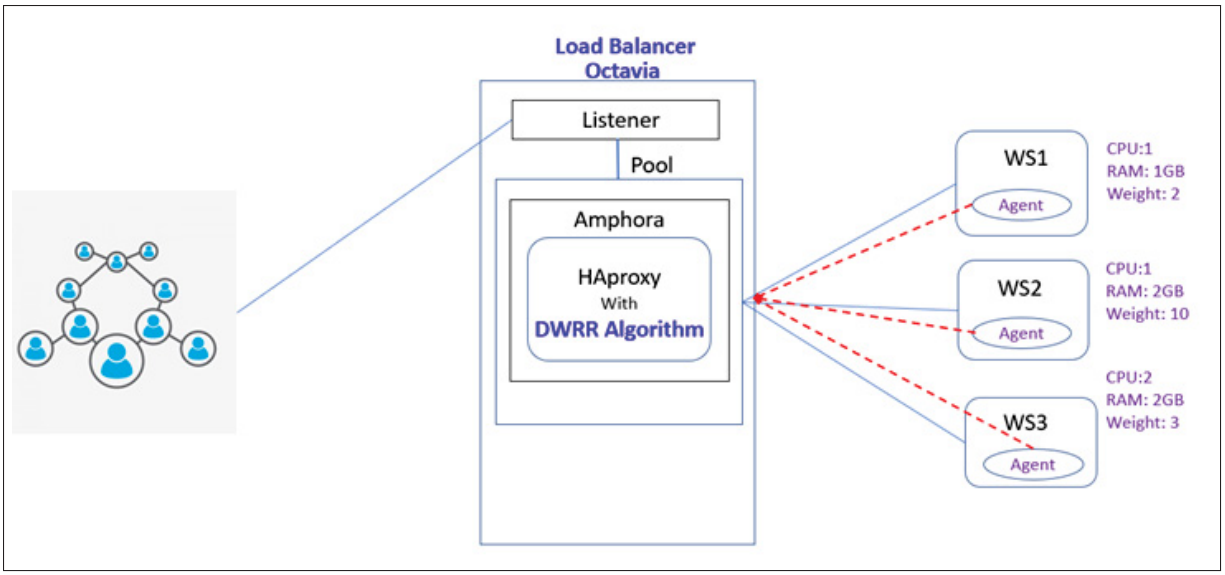

Figure 3.7 Load Balancer with Dynamic Version of WRR

Among different metrics related to load balancing algorithms in the literature review, CPU usage resulted in minimum response time and maximum resource utilization. Therefore, in our study, the percentage of CPU utilization for each web server is considered at run time to process the incoming requests. If the server is busy with a high CPU usage percentage, it will not process more requests. Otherwise, it can handle an increased number of requests.

Based on Figure 3.8, the agent includes two parts. One is located on the load balancer, with two threads. The main thread receives the information in real-time from web servers for calculating the new weight of each web server. The parallel one applies new web servers' weights based on power usage on backend nodes via load balancer. Therefore, the main thread is in charge of monitoring and processing the information coming from virtual machine instances. The worker thread applies new weights to instances through the load balancer. The second part of agent is placed on web servers to send the last status of virtual machine instances to load balancer every 1 second in the form of UDP packets using the port of 1366.

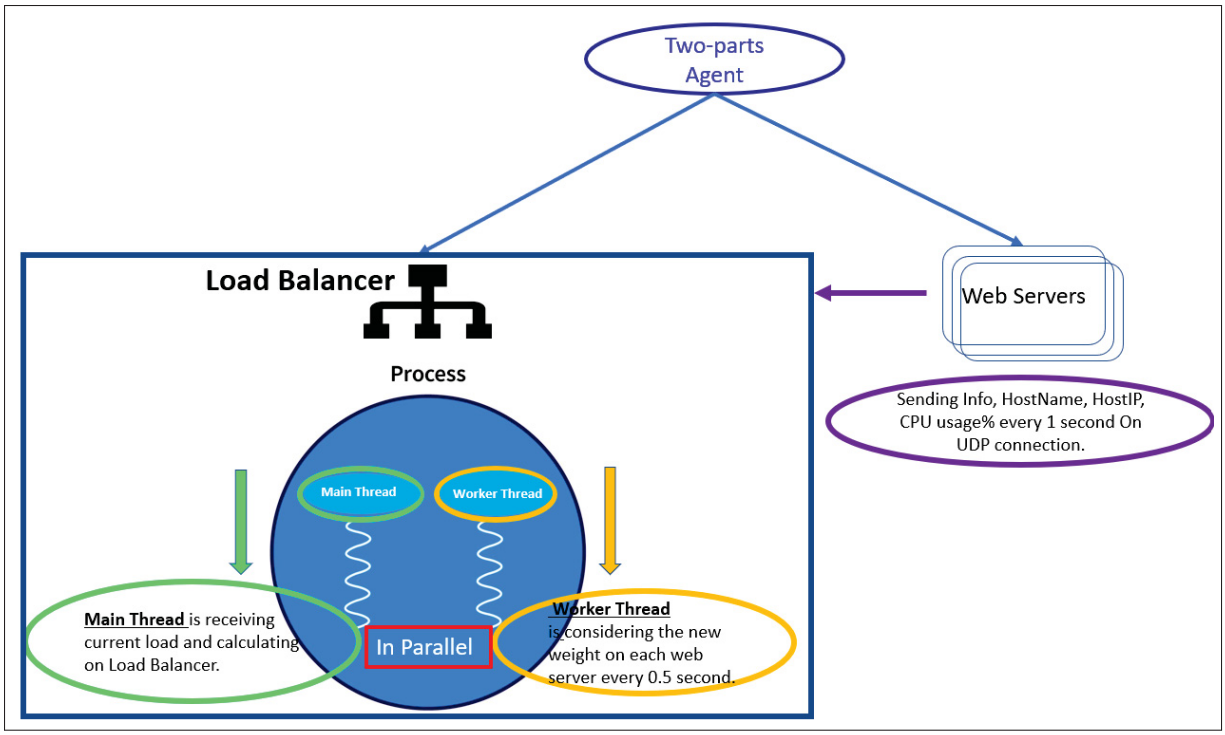

Figure 3.8 Two-parts Agent

Figure 3.9 depicts the flowchart of the agent-based load balancing solution with dynamic weight modification.

Based on this algorithm, new tasks are permanently assigned to the server with a low CPU percentage. If two servers have the same weight (same power usage), new requests are assigned randomly among them. By main thread, while new requests are coming from clients, the load balancer receives updated information about each web server, such as hostname, host IP and CPU percentage with ascending sort. Then, we store the hostnames and CPU percentage values separately to make the descending sort on the servers' weight values. Then, a new dictionary is created including keys (hostnames) and reversed values as new weights. In parallel, the worker thread updates new weights on the load balancer. Next, the new tasks are assigned to the light server with low weight, and it dynamically switches among the servers which is not busy and ready to process the new tasks. After updating, the last status of each web server in the load balancer is equalled to a null value to ensure whether all servers are available or not in the pool.

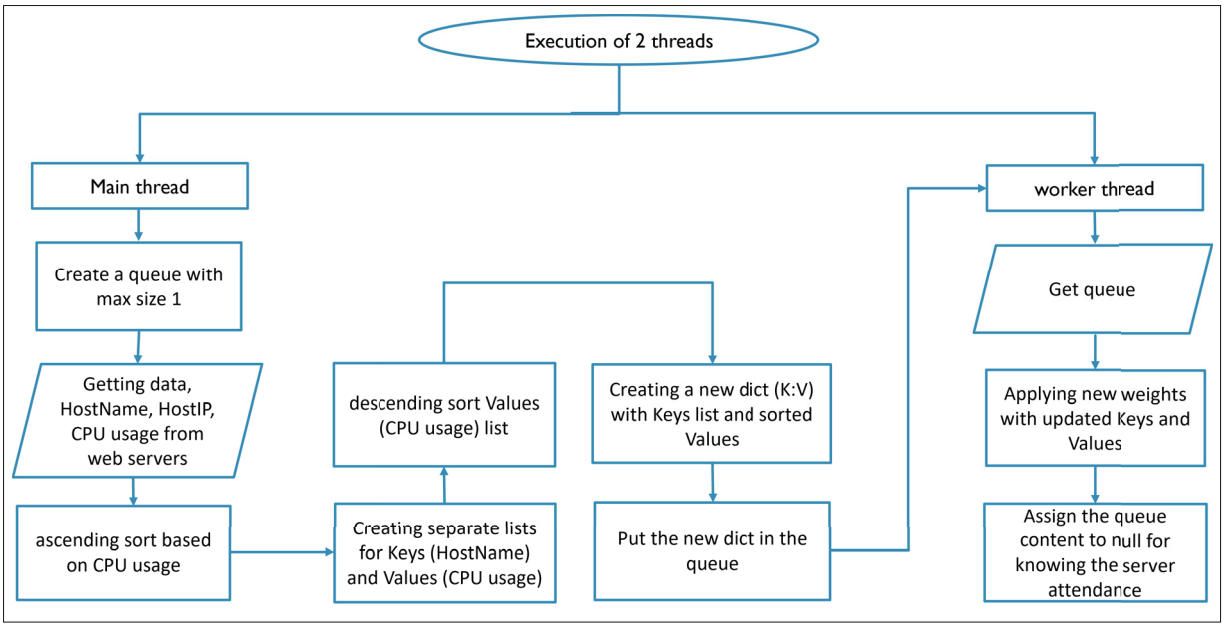

Figure 3.9 Dynamic Load balancing procedure

In this way, the load balancer finds the new status of web servers and will not consider the server which is down. This feature prevents requests from being lost and results in timely execution and better performance in terms of load balancing. Following that, this function continues until all requests are processed.

To show the difference between the static and dynamic methods in terms of the number of established connections, we use "tcpdump" for discovering network information (Wikipedia contributors, 2023).

According to Table 3.5, we use tcpdump as a data packet network analyzer to capture the traffic on each web server with BPF, Berkeley Packet Filters, (IBM, 2022). By this, we find the number of processed requests on servers.

Table 3.5 Capturing the number of requests on servers

tcpdump Analyzer	Switch	BPF Filter	port
	-n : to display name resolution of hostnames –i : to capture traffic on an interface	‘tcp[13]& 8!=0’	80

As we mentioned in Table 3.5, we utilize the BPF filter to capture all TCP-PSH packets to determine how many client requests were processed on each server after the three-way handshaking phase.

Figure 3.10, at a glance, describes how the load balancer calculates the current loads based on CPU percentage to generate the new weight for each web server. In the following, we will explain more about weight modification processing (current load, calculation load, and new load weight) based on the mentioned procedure of the dynamic version of WRR.

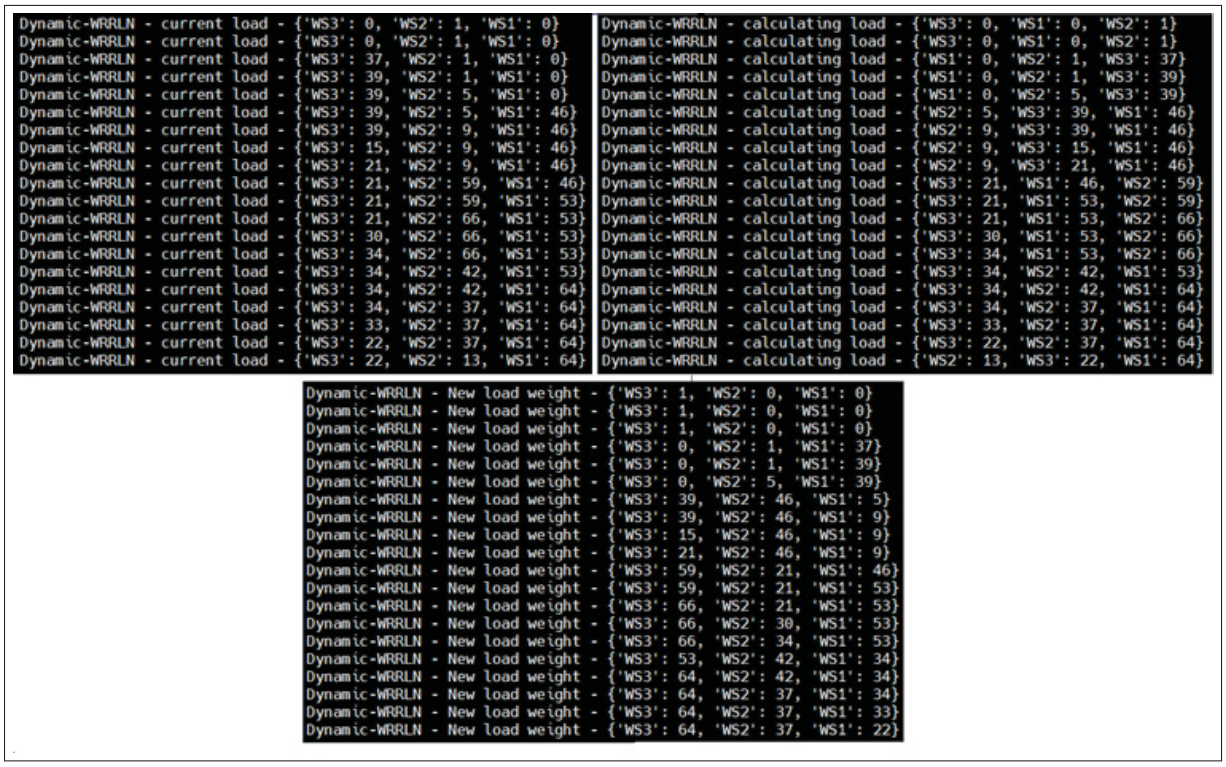

Figure 3.10 Logs for Dynamic version of WRR

In Figure 3.11 regarding the current load, each server presents a number that refers to each server's real-time CPU percentage. In other words, each server shows how much CPU is occupied for processing internal and external tasks as current load in real-time.

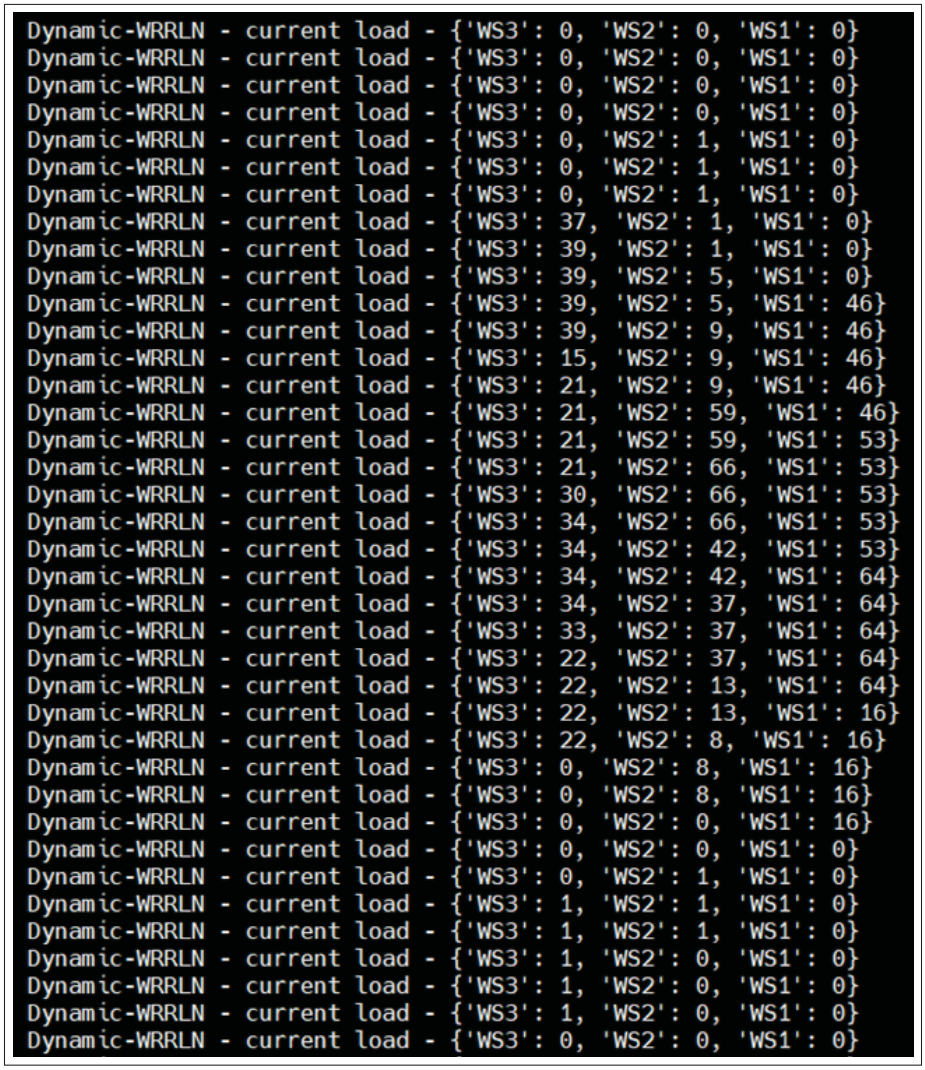

Figure 3.11 Logs of Current Load in Dynamic-WRR

In Figure 3.12, the current load is sorted in ascending order. For example, if the CPU percentage of each web server as the current load is WS3:39, WS2:5 and WS1:46, then after ascending sort, it will be WS2:5, WS3:39 and WS1:46.

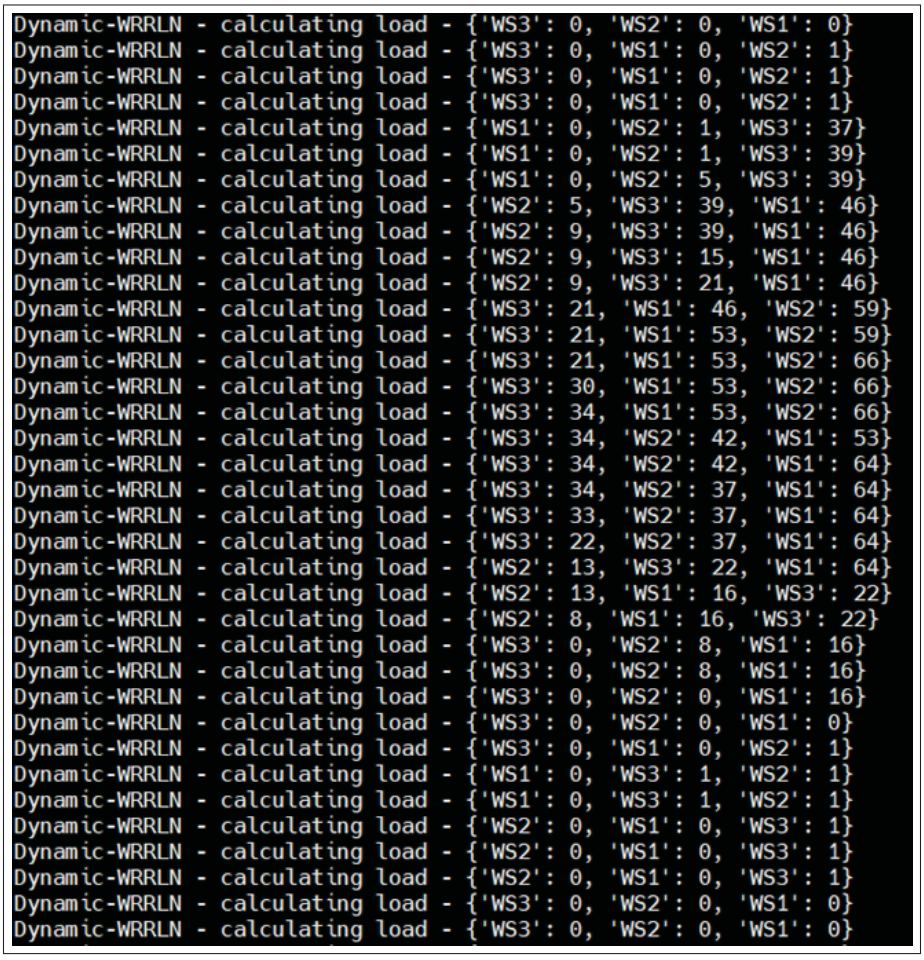

Figure 3.12 Calculating load's Log in Ascending Sort

In the following, Figure 3.13 shows the new weight calculated after Ascending sort. In this Figure, three steps are carried out on the ascending sorted current load to generate the new weight of each server. First is separating the Hostnames and CPU percentage values into two lists. Second, the CPU values are sorted in descending order. Then the Hostname lists and descending values are merged as new weights for the load balancer to distribute the new tasks. By this, weights are dynamically modified according to the current load of each server. This log shows that new requests go to the server with low power processing usage. For example, WS1, with 9% CPU usage, changes to 46%. Then it increases to 53% in the following process.

```
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 1, 'WS1': 37}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 1, 'WS1': 39}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 5, 'WS1': 39}
Dynamic-WRRLN - New load weight - {'WS3': 39, 'WS2': 46, 'WS1': 5}
Dynamic-WRRLN - New load weight - {'WS3': 39, 'WS2': 46, 'WS1': 9}
Dynamic-WRRLN - New load weight - {'WS3': 15, 'WS2': 46, 'WS1': 9}
Dynamic-WRRLN - New load weight - {'WS3': 21, 'WS2': 46, 'WS1': 9}
Dynamic-WRRLN - New load weight - {'WS3': 59, 'WS2': 21, 'WS1': 46}
Dynamic-WRRLN - New load weight - {'WS3': 59, 'WS2': 21, 'WS1': 53}
Dynamic-WRRLN - New load weight - {'WS3': 66, 'WS2': 21, 'WS1': 53}
Dynamic-WRRLN - New load weight - {'WS3': 66, 'WS2': 30, 'WS1': 53}
Dynamic-WRRLN - New load weight - {'WS3': 66, 'WS2': 34, 'WS1': 53}
Dynamic-WRRLN - New load weight - {'WS3': 53, 'WS2': 42, 'WS1': 34}
Dynamic-WRRLN - New load weight - {'WS3': 64, 'WS2': 42, 'WS1': 34}
Dynamic-WRRLN - New load weight - {'WS3': 64, 'WS2': 37, 'WS1': 34}
Dynamic-WRRLN - New load weight - {'WS3': 64, 'WS2': 37, 'WS1': 33}
Dynamic-WRRLN - New load weight - {'WS3': 64, 'WS2': 37, 'WS1': 22}
Dynamic-WRRLN - New load weight - {'WS3': 22, 'WS2': 64, 'WS1': 13}
Dynamic-WRRLN - New load weight - {'WS3': 13, 'WS2': 22, 'WS1': 16}
Dynamic-WRRLN - New load weight - {'WS3': 8, 'WS2': 22, 'WS1': 16}
Dynamic-WRRLN - New load weight - {'WS3': 16, 'WS2': 8, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 16, 'WS2': 8, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 16, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 1}
Dynamic-WRRLN - New load weight - {'WS3': 1, 'WS2': 0, 'WS1': 1}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 1, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 1, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 1, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 0, 'WS1': 0}
Dynamic-WRRLN - New load weight - {'WS3': 0, 'WS2': 0, 'WS1': 0}
```

Figure 3.13 New load weight logs in Dynamic-WRR

Table 3.6, comes from a part of actual results based on Figure 3.10 to see what exactly occurred via the dynamic version of WRR in the backend. With reading the table from left to right, the first column shows the current CPU usage percentage of each server. Then, in the second and third columns, after calculation, including ascending, descending and merging, a new weight is set for each server in the load balancer to assign new tasks. Therefore, the weight and the number of requests for each server with high CPU usage are decreased. Moreover, CPU usage percentage depends on internal and external tasks as well as Operating System updates.

Table 3.6 Calculation of current load

Step	Current load (CPU usage percentage)	Ascending	Descending & creating new weight
1	{WS3: 37, WS2: 1, WS1: 0}	{WS1: 0, WS2: 1, WS3: 37}	{WS3: 0, WS2: 1, WS1: 37}
2	{WS3: 39, WS2: 1, WS1: 0}	{WS1: 0, WS2: 1, WS3: 39}	{WS3: 0, WS2: 1, **WS1: 39**}
3	{WS3: 39 , WS2: 5 , WS1: 0}	{WS1: 0, WS2: 5, WS3: 39}	{WS3: 0 **WS2: 5**, WS1: 39}
4	{WS3: 39 , WS2: 5, WS1: 46}	{WS2: 5, WS3: 39, WS1: 46}	{WS3: 39, **WS2: 46**, **WS1: 5**}
5	{WS3: 39, WS2: 9, WS1: 46}	{WS2: 9, WS3: 39, WS1: 46}	{WS3: 39, WS2: 46, **WS1: 9**}
6	{WS3: 15, WS2: 9, WS1: 46}	{WS2: 9, WS3: 15, WS1: 46}	{**WS3: 15**, WS2: 46, WS1: 9}
7	{WS3: 21 , WS2: 9, WS1: 46}	{WS2: 9, WS3: 21, WS1: 46}	{**WS3: 21**, WS2: 46, WS1: 9}
8	{WS3: 21, WS2: 59, WS1: 46}	{WS3: 21, WS1: 46, WS2: 59}	{**WS3: 59**, **WS2: 21**, WS1: 46}
9	{WS3: 21, WS2: 59, WS1: 53}	{WS3: 21, WS1: 53, WS2: 59}	{WS3: 59, WS2: 21, **WS1: 53**}
10	{WS3: 21, WS2: 66, WS1: 53}	{WS3: 21 , WS1: 53, WS2: 66}	{**WS3: 66**, WS2: 21, WS1: 53}

3.6 Summary

This chapter explained how we installed and configured the OpenStack cloud environment with three nodes, Controller, Compute, and Network. In the Controller node, the essential modules for our experiment, such as Nova, Keystone, Glance, and Neutron, were installed and configured. In Compute node, we installed Nova-compute, one of the Nova components, Neutron-plugin-agent and QEMU hypervisor for hosting three web servers needed for our experiment. We also deployed the Octavia API in the Network node to launch Amphora as a load balancer service with an existing static load balancing method, WRR, and three members (WS1, WS2, and WS3) to balance client loads. For sending client requests to the load balancer, we simulated concurrent HTTP requests via multi threads on TCP socket. In this workload simulation, we estimated the time of request processing operation that will be used for evaluation in Chapter 4.

By performing the WRR algorithm, the logs showed the processing power usage of each web server in the static method in which one server is busier than others.

For the next step, we proposed our solution as an agent-based method upon the static method, WRR, and explained the dynamic load balancing procedure in real-time with an agent. The agent includes two parts. The first is located within the load balancer, including two threads in parallel. One is a main thread to receive the current load and calculate it. Another is a worker thread to update new weights. After applying each update on servers by the worker thread, the last status of each web server is assigned to a null value to check the servers' attendance. Second, another part of the agent is placed in each web server to send the last status of each server.

Then, we briefly described the logs based on servers' CPU usage in static and dynamic algorithms for balancing the tasks coming from the clients. Moreover, to count the number of processed clients' requests per server, we used a command-line packet network analyzer, tcpdump, to capture real-time network traffic.

CHAPTER 4

RESULTS AND DISCUSSION

After adding a dynamic version load balancing based on the WRR algorithm in a cloud environment via OpenStack as the thesis's main intention, we will evaluate the outcomes of the dynamic load balancer compared to the static one in this chapter. In addition to the main purpose of this thesis, we will compare the static and dynamic methods in choosing the lightest server for processing incoming requests based on the CPU utilization metric.

4.1 Network Topology

As mentioned in the previous chapter, we created the cloud environment via OpenStack with three VMs. We installed and configured the QEMU hypervisor in Compute VM to create three servers with Ubuntu 20.04 with different configurations based on Table 3.2. At first, These servers, WS1, WS2, and WS3, connect to a Local private Local network to communicate with each other. Then they connect to a Flat provider network through a router connecting to an external network. Load balancer within the Amphora located in Compute node to receive the load via floating IP on UDP connection due to prevent latency.

To make web servers, we logged in to each server via keypair for installing Apache2. The network loads are simulated with HTTP requests through multi threads sending to Load balancer.

A Python script is used as an agent and consists of two parts. One is on each web server to send real-time status to the load balancer, including hostname, host IP, and CPU usage percentage. Another part of the agent is on the load balancer to receive the information from the webserver-side agent to distribute the loads on each web server based on the weight considered in the load balancer configuration.

4.2 Experiment and Results Analysis

For our test environment configuration, we consider different hardware configurations and numeric weights for the web servers based on Table 3.6. To apply loads on web servers, we deploy concurrent HTTP requests to each of the load balancers, static and dynamic, ranging from 1000 to 20000. To generate the requests, as we mentioned in the previous chapter, the Python script is created with a socket library and multi-threads to produce the concurrent client requests.

As a load balancer, Octavia receives all the requests via Floating IP on the listener and then transfers them to the servers' pool based on the defined algorithm considering CPU utilization metric.

4.3 Weighted Round-Robin Algorithm

For the static method, to verify the effectiveness of this algorithm, we set the different configurations for the provider servers WS1, WS2, and WS3, considering different weights with fixed values 2, 10 and 3. We perform minimum and maximum simulation concurrent requests for this experiment, 1K to 20K, respectively. In this scenario, we consider different weights regardless of resource capabilities to see the outcomes. For the whole execution, the distribution is based on the initial weight without adjustment.

In this method, the responses to requests for each web server differ according to the weight. Based on Figure 4.1, the number of requests for WS2 with weight 10 is much more than WS1 and WS3 with weight 2 and 3, respectively. It results in a significant difference between the servers' weights. In other words, WS2 processed more requests than WS1 and WS3 due to maximum weight. Therefore, the CPU usage percentage of provider WS2 is much more than other providers' CPU utilization amount, according to logs in Figure 3.6.

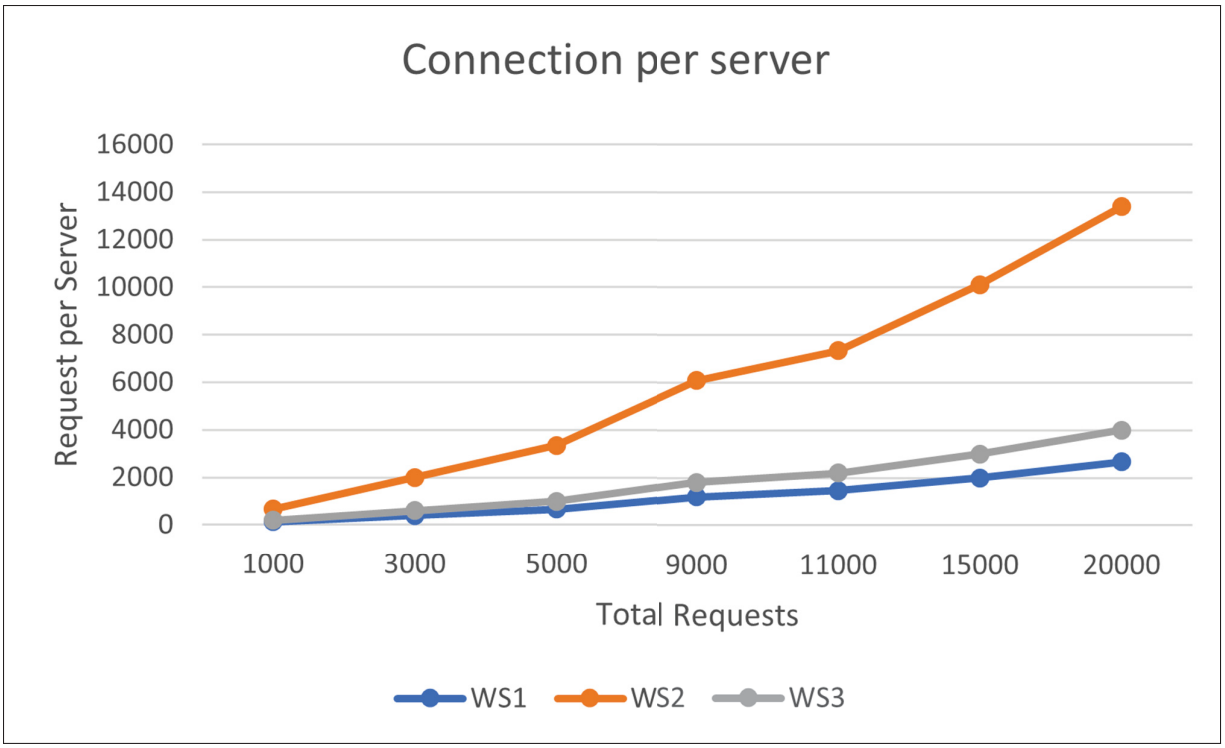

Figure 4.1 Static Load Balancing

4.4 Dynamic (Agent-based) Weighted Round-Robin Algorithm

The second version to be benchmarked is Dynamic WRR with the same configuration we set in the previous method. In this version, the balancer takes into consideration the last status of each server to modify the weight. The values such as Hostname, Host IP and CPU usage are received from the instances through the agent and calculated in the balancer to choose the next server. Compared with the static method, Figure 4.2 shows no considerable difference between the number of processed requests of each VM due to adjusting weight by the DWRR algorithm based on the real-time status of instances. Therefore, the servers in this method can handle the requests with better performance.

In this experiment, we execute a maximum of 20000 concurrent simulation requests on the weighted Round-Robin algorithm and agent-based method with dynamic weight adjustment.

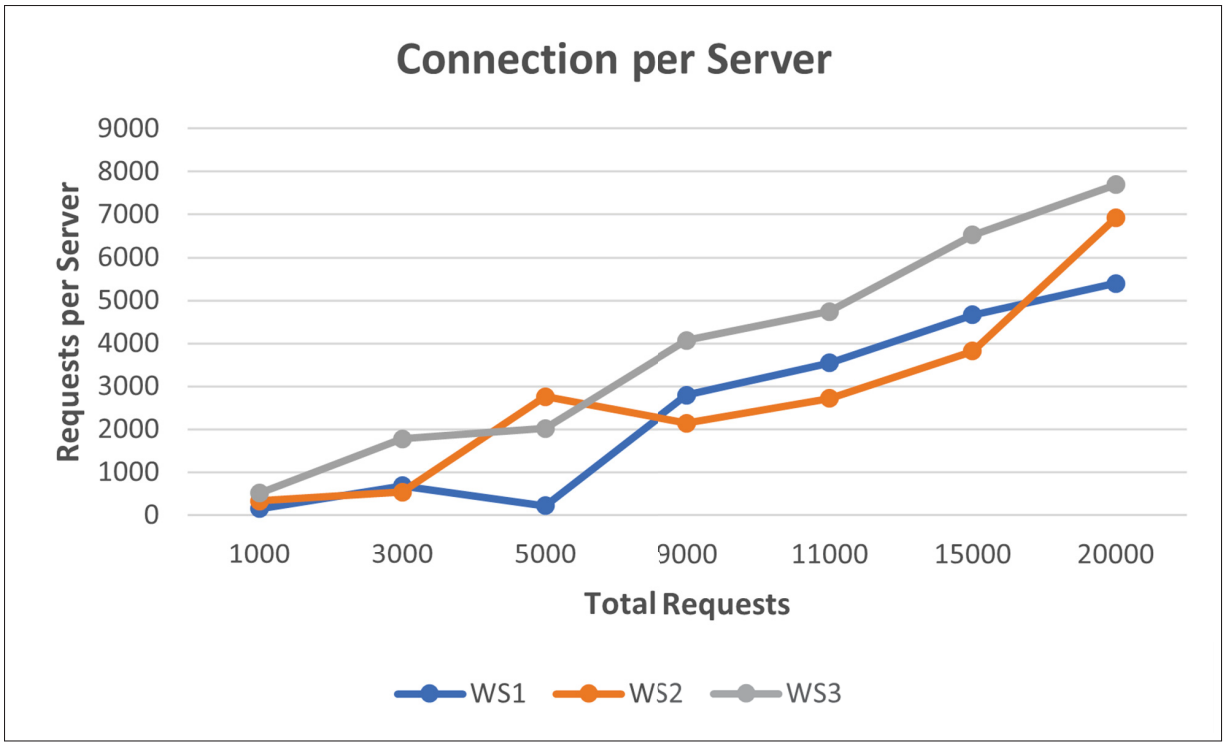

Figure 4.2 Dynamic Load Balancing

The following tables show the number of requests distributed among providers WS1, WS2, and WS3. Table 4.1 illustrates the number of requests on each server in dynamic mode, and Table 4.2 describes them in the static method.

Table 4.1 Connection Counts for Dynamic WRR

The number of connections per server (Dynamic mode)							
Device	**1000**	**3000**	**5000**	**9000**	**11000**	**15000**	**20000**
WS1	154	687	223	2790	3541	4662	5393
WS2	333	534	2758	2143	2719	3819	6917
WS3	513	1779	2019	4067	4740	6519	7690

Table 4.2 Connection Counts for Static WRR

The number of connections per server (Static mode)							
Device	**1000**	**3000**	**5000**	**9000**	**11000**	**15000**	**20000**
WS1	133	400	666	1200	1466	2000	2666
WS2	667	2000	3334	6000	7334	10000	13361
WS3	200	600	1000	1800	2200	3000	4000

In Figure 4.1, with static mode, tasks are distributed inappropriately regardless of the resource configuration. Consequently, It shows inappropriate gaps if weight values are incompatible with the servers' resources. Whereas, in the dynamic method, after considering the initial weight value at the first step, the weight of servers is continuously adjusted according to the resource configuration of the servers.

If the weights are chosen inappropriately, the static algorithm works blindly to assign the tasks among servers compared to the dynamic one, which constantly monitors the real-time status.

When the test is carried out, we record the average processing response time per server for static and dynamic methods, as in Figure 4.3.

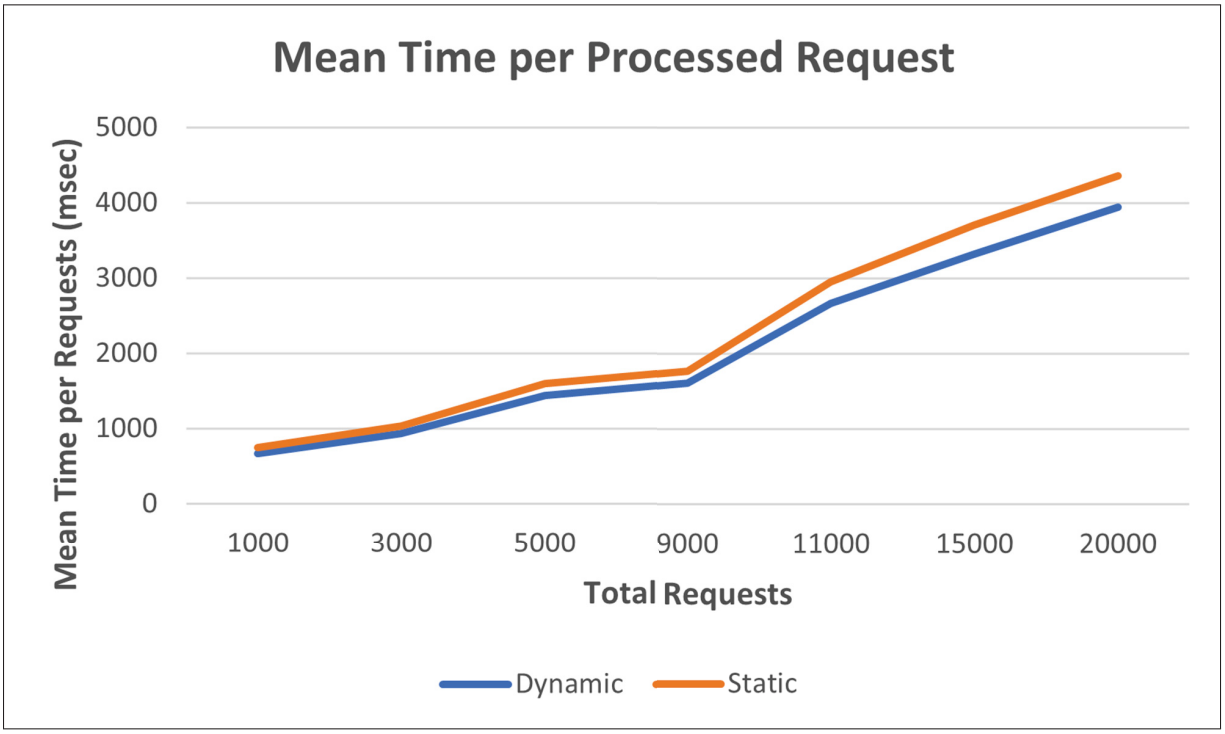

Figure 4.3 Request Completion Time on Average between Static and Dynamic Load Balancers

Figure 4.3 depicts that in the DWRR algorithm, the average response time is less than the mean time in the WRR algorithm due to the dynamic weight reflecting the real-time load status. Graph

4.3 shows that DWRR algorithm considering CPU usage metric provides better performance with approximately 10% efficiency.

4.5 Summary

After preparing the cloud environment in the OpenStack platform, we deployed one of the static load balancing methods, WRR, within the Octavia module. Next, we sent simulated concurrent HTTP requests ranging from 1k to 20k via multi threads like an actual situation to load balancer. Then, we observed the web servers' logs based on different CPU usage percentages, with a big gap between them.

In the static method, the loads were distributed based on initial numeric weight, which refers to CPU utilization percentage. In this method, each server with maximum weight received more requests than the others without considering the hardware utilization. Therefore, a server with maximum weight had a much more CPU percentage than other servers. Thus, one server was always busier than others because there was no information about servers' last status.

Our proposed solution was implementing the dynamic version of Weighted Round Robin algorithm (WRR) in this thesis. The DWRR algorithm estimated the web servers' current load status to adjust the servers' weight automatically based on the real-time hardware utilization measure such as CPU usage. Therefore, according to the outcomes, there was less gap in the number of requests per server compared to the static one, which resulted in small load fluctuations among web servers' CPU usage percentage. By this, the average response time per server's request was decreased, and each server could be fully utilized in hardware configuration compared to the static method.

Results showed that the dynamic version of WRR based on processing power usage has better performance with around 10% efficiency.

CONCLUSION AND RECOMMENDATIONS

According to the rapid progress and development in IT technology, Cloud computing is one of the most usable distributed systems, service-oriented with various service models and low cost in executing expenses. As users have high demands to access different online services by cloud providers on the Internet, load balancer is needed to allocate workloads among providers and clients' requests. Load balancer in Cloud in software mode via SDN is better than hardware in terms of cost and configuration. Different techniques in load balancing, static and dynamic, are reviewed in the related works. The literature review focuses on dynamic models considering high performance, fault tolerance, and low operation time. However, load balancing is the main issue in Cloud that has not been completely addressed. To the best of our knowledge, those dynamic mechanisms have not been deployed based on the static method in the OpenStack platform, one of the most popular open-source cloud platforms. In other words, as far as we know, there is no dynamic solution based on the static load balancing method in the OpenStack platform.

Therefore, we successfully implemented an agent in load balancing based on CPU usage metric over the static method, Weighted Round Robin, in a cloud environment via OpenStack on VirtualBox hypervisor to choose the next web server. Then we compared the static and dynamic mechanisms to determine a proper server to process the tasks based on CPU utilization.

The experiment was tested in a scenario with three configuration servers with various numeric weights to receive HTTP requests. Our solution was built on two threads through Python. The main thread is for receiving the last status of the servers' information to calculate the new weight of each server. The parallel thread is for updating the new weights via load balancer. From the results, we can conclude that our proposed algorithm, DWRR, allocates the workload better than WRR, the static algorithm. In DWRR as our solution, since the servers' weight automatically changes based on real-time load status, we observed less response time with

enhanced performance around 10% compared to the WRR algorithm showed in chapter 4, graph 4.3. Moreover, the results showed that almost all servers receive approximately the same number of requests in the dynamic version compared to the static one.

As future work, according to the findings of this study, three future works can be investigated. First is employing more than one metric to measure the current load within our solution to balance the workload dynamically in OpenStack platform to make higher efficiency with more than 10% observed in our study. Second is using Cloud bursting via OpenStack for dynamic load balancing that empowers companies to assign the required resources to multiple cloud environments in different geographical zones to guarantee optimal performance and workload balance. Last is using Machine Learning to track different loads, such as CPU, Memory, and Network, in real time to define and assign appropriate weights among backend web servers to balance the users' requests through agent-based load balancing within Octavia in the OpenStack platform. In this scenario, also scaling out the resources can be considered using Machine Learning to track the status of virtual machine instances in peak period time and make the best decision, such as provisioning a new virtual machine instance with the same configuration as a web server to serve the clients' requests.

APPENDIX I

OPENSTACK ARCHITECTURE

0.1 OpenStack Components

To build a cloud environment for this project via OpenStack, some infrastructure components as requirements and core components to provide services such as identity, compute, image, and network are installed and configured to meet the needs of the scenario, which is mentioned in chapter three based on the following Figure.

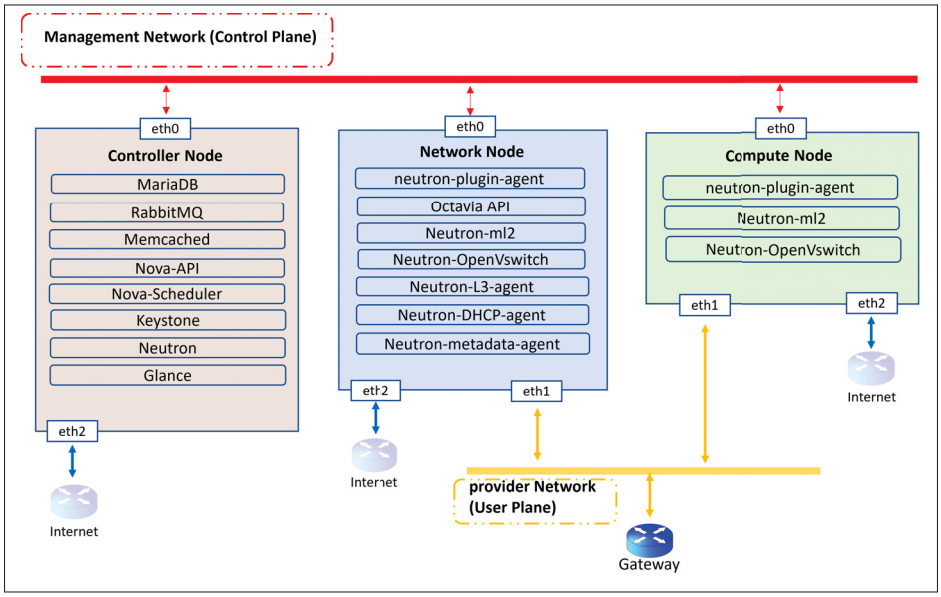

Figure-A I-1 Network Architecture

1. Infrustructure Components

To build the cloud environment via OpenStack, the following infrustructre components installed and configured in advanced.

- **MariaDB**
- **RabbitMQ**
- **Memcached**

1.1 MariaDB

To store information, OpenStack supports MariaDB or MySQL, and other SQL databases (OpenStack, 2023a). In this project MariaDB is used for core components' database located on Controller node.

1.2 RabbitMQ

In OpenStack, different message queue services are used, such as RabbitMQ, Qpid, and ZeroMQ, for coordinating operations among services (OpenStack, 2023c). In this project, RabbitMQ is located on Controller node to send requests to related agents and components to run the load balancer and network services to serve the users' requests. Therefore, core components can communicate with their related agents through the RabbitMQ service.

1.3 Memcached

Memcached is used by OpenStack core components to store temporary data such as tokens (OpenStack, 2023b). In this project, Memcached is located in the Controller node to cache tokens for improving the overall performance of the cloud environment.

2. Core Components

To create a cloud architecture for this project with three virtual nodes, Controller, Network, and Compute via OpenStack, the essential components such as Keystone, Glance, Nova, Neutron, and Octavia are needed to provide application load balancing on users' requests considering one data center. More explanation is as follows:

- **Keystone**
- **Glance**
- **Nova**
- **Neutron**
- **Octavia**

2.1 Keystone

Keystone, as an identity service in the Controller node, provides centralized authentication and authorization using username and password for other OpenStack components such as Glance, Nova, Neutron, and Octavia to access resources and execute requests to provide web services in this project.

2.2 Glance

Glance is located in the Controller node and played a role as an image service for virtual machine instances (Web Servers) and is used to store the operating system (Ubuntu20.04) images for each web server in this project.

2.3 Nova

Nova, as the control unit of OpenStack is responsible for managing instances and correlation with other components for providing identity, image, network, and load balancer services using Octavia in OpenStack to reach the goal in this thesis. Nova, located in the Controller node, comprises some sub-components, Nova-API, Nov-compute, Nova-scheduler, and Nova-conductor, for its tasks. Nova-API receives all the requests from other components for responding to them. Nova-Compute is for managing backend web servers within compute node. Nova-scheduler is for connecting to the compute node (in this project, just one compute node is considered for hosting the virtual machine web servers). And Nova-conductor is used to update web servers' database on Compute node.

2.4 Neutron

In this project, one of the other core components is Neutron, OpenStack networking as network services. Neutron and its components are deployed to connect the virtual machine instances as web servers to the external network for serving the clients' requests. These sub-components are Neutron-API for receiving the network requests, ML2 Core plugin to forward the requests to the

OVS agents located in other nodes (Network and Compute nodes in this project are configured in separate Virtual machine nodes for user data plane), and OVS switch which is configured through OVS agent, as well as DHCP agent to configure DHCP server within network node to assign IP addresses for virtual machines as web servers.

2.5 Octavia

As another main component for this project, Octavia is used to act as a load balancer based on DWRR algorithm as a dynamic solution to spread incoming HTTP requests between designated instaces as backend web servers. Octavia API and its sub-components, such as the Controller worker, Health Manager, Housekeeping Manager, and Driver Agent, are placed in the network node in this project.

BIBLIOGRAPHY

Al-Mashhadi, S., Anbar, M., Jalal, R. A. & Al-Ani, A. (2020). Design of Cloud Computing Load Balance System Based on SDN Technology. 603, 123–133. doi: https://doi.org/10.1007/978-981-15-0058-9_13. Series Title: Lecture Notes in Electrical Engineering.

Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J. & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6), 599–616. doi: https://doi.org/10.1016/j.future.2008.12.001.

Callegati, F., Cerroni, W., Contoli, C. & Santandrea, G. (2014). Performance of Network Virtualization in cloud computing infrastructures: The OpenStack case. 132–137. doi: https://doi.org/10.1109/CloudNet.2014.6968981.

Dasgupta, K., Mandal, B., Dutta, P., Mandal, J. K. & Dam, S. (2013). A Genetic Algorithm (GA) based Load Balancing Strategy for Cloud Computing. *Procedia Technology*, 10, 340–347. doi: https://doi.org/10.1016/j.protcy.2013.12.369.

Denton, J. (2016). *OpenStack Networking Essentials*. Packt Publishing Ltd.

HAProxy. (2023). HAProxy - The Reliable, High Perf. TCP/HTTP Load Balancer. Retrieved on 2023-06-18 from: https://www.haproxy.org/.

Harvey. (2014). 60 Open Source Apps You Can Use in the Cloud | Datamation. Retrieved on 2023-06-18 from: https://www.datamation.com/open-source/60-open-source-apps-you-can-use-in-the-cloud/.

Hu, J., Gu, J., Sun, G. & Zhao, T. (2010). A Scheduling Strategy on Load Balancing of Virtual Machine Resources in Cloud Computing Environment. *2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming*, pp. 89-96. Retrieved from: https://ieeexplore.ieee.org/document/5715067.

Huawei Technologies Co., Ltd. (2023). *Cloud Computing Technology*. Singapore: Springer Nature Singapore. doi: https://doi.org/10.1007/978-981-19-3026-3.

Huo, J., Qu, H. & Wu, L. (2015, 12). Design and Implementation of Private Cloud Storage Platform Based on OpenStack. pp. 1098-1101. Retrieved from: https://ieeexplore.ieee.org/document/7463870.

IBM. (2022). IBM Documentation. Retrieved on 2023-02-20 from: https://ibm.com/docs/en/qsip/7.4?topic=queries-berkeley-packet-filters.

Kulkarni, V., Aldi, S. S., Mulla, M. M., Narayan, D. G. & Hiremath, P. S. (2022). Dynamic Live VM Migration Mechanism in OpenStack-Based Cloud. *2022 International Conference on Computer Communication and Informatics (ICCCI)*, pp. 1–6. Retrieved from: https://ieeexplore.ieee.org/document/9740780.

Kumar, S. & Rana, D. S. (2015). Various Dynamic Load Balancing Algorithms in Cloud Environment: A Survey. *International Journal of Computer Applications*, 129, 14-19. Retrieved from: https://www.researchgate.net/publication/284223774_Various_Dynamic_Load_Balancing_Algorithms_in_Cloud_Environment_A_Survey.

Li, D., Zheng, Z., Li, Y., Xu, Y. & Tang, D. (2017). Design and Implementation of Load Balancing Strategy in Openstack Cloud Platform. *22017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, pp. 428–435. Retrieved from: http://ieeexplore.ieee.org/document/8005835/.

Mell, P. M. & Grance, T. (2011). The NIST Definition of Cloud Computing. *NIST*. doi: https://doi.org/10.6028/NIST.SP.800-145. Last Modified: 2018-11-10T10:11-05:00 Publisher: Peter M. Mell, Timothy Grance.

Mishra, S. K., Sahoo, B. & Parida, P. P. (2020). Load balancing in cloud computing: A big picture. *Journal of King Saud University - Computer and Information Sciences*, 32(2), 149–158. doi: https://doi.org/10.1016/j.jksuci.2018.01.003.

Moganarangan, N., Babukarthik, R. G., Bhuvaneswari, S., Basha, M. S. S. & Dhavachelvan, P. (2016). A novel algorithm for reducing energy-consumption in cloud computing environment: Web service computing approach. *Journal of King Saud University - Computer and Information Sciences*, 28(1), 55–67. doi: https://doi.org/10.1016/j.jksuci.2014.04.007.

Nakrani, S. & Tovey, C. (2004). On Honey Bees and Dynamic Server Allocation in Internet Hosting Centers. *Adaptive Behavior*, 12(3-4), 223-240. doi: https://https://doi.org/10.1177/105971230401200308.

OpenStack. (2018a). Additional networking services — arch-design 0.0.1.dev15 documentation. Retrieved on 2023-06-17 from: https://docs.openstack.org/arch-design/design-networking/design-networking-services.html#lbaas.

OpenStack. (2018b). Design — arch-design 0.0.1.dev15 documentation. Retrieved on 2023-06-18 from: https://docs.openstack.org/arch-design/design.html.

OpenStack. (2021). OpenStack Docs: System architecture. Retrieved on 2023-06-18 from: https://docs.openstack.org/nova/pike/admin/arch.html.

OpenStack. (2023a). OpenStack Docs: SQL database. Retrieved on 2023-07-30 from: https://docs.openstack.org/mitaka/install-guide-rdo/environment-sql-database.html.

OpenStack. (2023b). Memcached — Installation Guide documentation. Retrieved on 2023-07-30 from: https://docs.openstack.org/install-guide/environment-memcached.html.

OpenStack. (2023c). OpenStack Docs: Message queue. Retrieved on 2023-07-30 from: https://docs.openstack.org/mitaka/install-guide-ubuntu/environment-messaging.html.

OpenStack. (2023d). Introducing Octavia — octavia 11.1.0.dev42 documentation. Retrieved on 2023-02-01 from: https://docs.openstack.org/octavia/latest/reference/introduction.html.

OpenStack. (2023e). Networking — Security Guide documentation. Retrieved on 2023-06-18 from: https://docs.openstack.org/security-guide/networking.html.

OpenStack. (2023f). Networking services — Security Guide documentation. Retrieved on 2023-06-18 from: https://docs.openstack.org/security-guide/networking/services.html.

OpenStack. (2023g). Neutron/LBaaS/Deprecation - OpenStack. Retrieved on 2023-02-01 from: https://wiki.openstack.org/wiki/Neutron/LBaaS/Deprecation#Why_are_we_deprecating_neutron-lbaas.3F.

OpenStack. (2023h). OpenStack Docs: Load Balancer as a Service (LBaaS). Retrieved on 2023-01-24 from: https://docs.openstack.org/neutron/rocky/admin/config-lbaas.html.

OpenStack. (2023i). Introduction to OpenStack — Security Guide documentation. Retrieved on 2023-06-18 from: https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html.

Randles, M., Lamb, D. & Taleb-Bendiab, A. (2010). A Comparative Study into Distributed Load Balancing Algorithms for Cloud Computing. *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pp. 551–556. Retrieved from: https://ieeexplore.ieee.org/document/5480636.

Rimal, B. P., Choi, E. & Lumb, I. (2009). A Taxonomy and Survey of Cloud Computing Systems. *2009 Fifth International Joint Conference on INC, IMS and IDC*, pp. 44-51. Retrieved from: https://ieeexplore.ieee.org/document/5331755.

Rista, A., Ajdari, J. & Zenuni, X. (2020). Cloud Computing Virtualization: A Comprehensive Survey. *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*, pp. 462–472. Retrieved from: https://ieeexplore.ieee.org/document/9245124/.

Rong He, Xinming Tan. (2018). A load balancing algorithm with dynamic adjustment of weight. *Proceedings of 2018 the 8th International Workshop on Computer Science and Engineering*. doi: https://10.18178/wcse.2018.06.110.

Singh, A., Juneja, D. & Malhotra, M. (2015). Autonomous Agent Based Load Balancing Algorithm in Cloud Computing. *Procedia Computer Science*, 45, 832–841. doi: https://doi.org/10.1016/j.procs.2015.03.168.

Swarnkar, N., Singh, A. P. A. K. & Shankar, D. R. (2013). A Survey of Load Balancing Techniques in Cloud Computing. *International Journal of Engineering Research & Technology*, 2(8). Retrieved from: https://www.ijert.org/a-survey-of-load-balancing-techniques-in-cloud-computing. Publisher: IJERT-International Journal of Engineering Research & Technology.

Tsai, C.-W. & Rodrigues, J. J. P. C. (2014). Metaheuristic Scheduling for Cloud: A Survey. *IEEE Systems Journal*, 8(1), 279-291. Retrieved from: https://ieeexplore.ieee.org/document/6516911.

Tumkur, G. (2016). Load Balancing As A Service In Openstack-Liberty. Retrieved from: https://www.researchgate.net/publication/327052299_Load_Balancing_As_A_Service_In_Openstack-Liberty.

Verma, D. (2017). *SOFTWARE DEFINED LOAD BALANCING OVER AN OPENFLOW-ENABLED NETWORK*. (Thesis). Retrieved from: http://hdl.handle.net/10106/26824.

VMware. (2023). What is Software-Defined Networking (SDN)? | VMware Glossary. Retrieved on 2023-06-17 from: https://www.vmware.com/topics/glossary/content/software-defined-networking.html.

Wikipedia contributors. [[Online; accessed 20-June-2023]]. (2023). Tcpdump — Wikipedia, The Free Encyclopedia. Retrieved from: https://en.wikipedia.org/w/index.php?title=Tcpdump&oldid=1157800909.

Wu, Y., Luo, S. & Li, Q. (2013). An Adaptive Weighted Least-Load Balancing Algorithm Based on Server Cluster. *2013 5th International Conference on Intelligent Human-Machine Systems and Cybernetics*, 1, 224–227. Retrieved from: https://ieeexplore.ieee.org/document/6643872.