Republic of Iraq
Ministry of Higher Education and Scientific Research
University of Babylon
College of Information Technology
Department of Information Networks

Enhancing Load Balancing Algorithms for Improving the Performance of Web Server

A Thesis
Submitted to the Council of the College of Information Technology for Postgraduate Studies of University of Babylon in Partial Fulfillment of the Requirements for the Degree of Master in Information Technology / Information Networks

By

Karar Haider Anun Mousa

Supervised by

Asst. Prof. Dr. Mahdi Salih Neama Almhanna

2022 A.D 1443 A.H

بسم الله الرحمن الرحيم

{عَلَّمَ الْإِنسَانَ مَا لَمْ يَعْلَمْ}

[سورة العلق/ الاية: 5]

صدق الله العلي العضيم

Declaration

I hereby declare that this thesis, entitled "**Enhancing Load Balancing Algorithms for Improving the Performance of Web Server**", submitted to Department of Information Networks / College of Information Technology / University of Babylon, as fulfillment of requirements for the degree of **Master in Information Technology - Information Networks**, has not been submitted as an exercise for a similar degree at any other university. I also certify that the work described here is entirely my own.

Signature:

Name: **Karar Haider Anun**

Date: / / 2022

Supervisor Certification

I certify that this thesis entitled "**Enhancing Load Balancing Algorithms for Improving the Performance of Web Server**" was prepared under my supervision at the Department of Information Networks / College of Information Technology / University of Babylon, by **Karar Haider Anun** as a partial fulfillment of the requirements for the degree of **Master in Information Technology - Information Networks.**

Signature:

Supervisor Name: **Asst. Prof. Dr. Mahdi Salih Neama Almhanna**

Date: / / 2022

Head of the Department Certification In view of the available recommendation, I forward this thesis entitled "**Enhancing Load Balancing Algorithms for Improving the Performance of Web Server**" for debate by the examination committee.

Signature:

Name: **Prof. Dr. Saad Talib Hasson**

Head of Information Networks Department

Date: / / 2022

Certification of the Examination Committee

We, the undersigned, certify that (**Karar Haider Anun**) candidate for the degree of Master in Information Technology-Information Networks, has presented his dissertation of the following title (**Enhancing Load Balancing Algorithms for Improving the Performance of Web Server**) as it appears on the title page and front cover of the dissertation that the said dissertation is acceptable in form and content and displays a satisfactory knowledge of the field of study as demonstrated by the candidate through an oral examination held on (19 / 5 / 2022) in our opinion, it is adequate with (**Very Good**) standing as a thesis.

Signature:
Name: Dr. Haydar Abdulameer Marhoon
Title: Asst. Prof.
Date: / 6 / 2022
(Chairman)

Signature:
Name: Dr. Ali Kadhum Al-Qurabat
Title: Asst. Prof.
Date: / 6 / 2022
(Member)

Signature:
Name: Dr. Mohammad Hussein Jawwad
Title: Asst. Prof.
Date: / 6 / 2022
(Member)

Signature:
Name: Dr. Mahdi Salih Neama
Title: Asst. Prof.
Date: / 6 / 2022
(Member and Supervisor)

Approved by the Dean of the College of Information Technology, University of Babylon.

Signature:
Name: Dr. Hussein Atiya Lafta
Title: Prof.
Date: / 6 / 2022
(Dean of College of Information Technology)

Acknowledgment

I would like to express my deepest gratitude to my supervisor, **Dr. Mahdi Salih Neama Almhanna,** for having provided valuable advice, motivation, guidance, and so many fruitful discussions throughout the preparation of this thesis.

Also, I would like to extend my respect and deepest gratitude to the College of Information Technology at the University of Babylon.

A special thanks to **Dr. Raaid Alubady**. Your advice on the thesis was great.

I would like to express my thanks to **My Mother** for deserving all her thanks for the effort and time she put into supporting me in completing my master's degree.

I would also like to express my thanks to my Uncle, my Mom's brother, **Ammar Al-Tahan**, for supporting me.

Sincere appreciation and love go to my family and friends. Finally, sincere thanks and appreciation go to every person who has an impact on continuing and completing this research. Thank you to everyone who encouraged me with at least one word.

Abstract

The process of dividing up network traffic amongst several different servers is referred to as load balancing. This ensures that not a single server is overloaded with requests. The much more evenly distributed the workload becomes, the better the application's responsiveness will be. Additionally, it expands the selection of websites and applications that consumers can access. Load balancers are necessary for modern apps to function properly.

The early pioneers of computing and the internet discovered that there are physical limits to the number of requests that can be processed in a given length of time by a computer. Fortunately, it appears that these physical boundaries are expanding at an exponential rate. However, the public's desire for speedy and complex software is continually stretching the boundaries of machines, since we are stacking hundreds of thousands or even millions of people onto them.

So businesses have two options available to use for solutions when confronted with increasing demand from users and reaching the performance limit of the server that is hosting your service: scale up or scale out. There are physical computing limitations when scaling up (also known as vertical scaling). Scaling out, also known as horizontal scaling, enables you to disperse the request load across as many servers as are required to handle the workload in a horizontal fashion. Using a load balancer can help spread the requests across a larger number of servers when scaling up.

The first part of the proposed system uses the load balancing algorithms (Round Robin (RR) and Least Connection (LC)) and enhances these algorithms because they have drawbacks. The proposed enhancement algorithms add weight to each server to achieve a boost to become the first algorithm Enhance Weight Round Robin (EWRR) and the second algorithm Enhance Weight Least Connection (EWLC) with backend programming for websites using Node JS as Default (Single-Thread). The second part of the proposed system is to improve performance, use Multi-Thread at Node JS for cluster request processing, and compare the algorithm's results.

The results of first part of the proposed system use Single-Thread with (RR) is 16k Requests and (LC) is 25k Requests and (EWRR) is 20k Requests and (EWLC) is 54k Requests. The results of second part of the proposed system use Multi-Thread with (RR) is 17k Requests and (LC) is 87k Requests and (EWRR) is 24k Requests and (EWLC) is 96k Requests.

Table of Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

API	*Application Programming Interface*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
Dev	*Development*
DevOps	*Developers And Operations*
DIRT	*Data-Intensive Real-Time Applications*
DWS	*Distributed Web Server*
HTTP	*Hypertext Transfer Protocol*
I/O	*Input/output*
IP	*Internet Protocol*
IT	*Information Technology*
JS	*JavaScript*
JSON	*JavaScript Object Notation*
JWT	*JSON Web Token*
LC	*Least Connection*
MB	*Megabyte*
ms	*Millisecond*
NCMC	*National Communications and Media Commission*
NoSQL	*Non-SQL*
NPM	*Node Package Manager*
PC	*Personal Computer*
QA	*Quality Assurance*

RAM	*Random Access Memory*
RDBMS	*Relational Database Management System*
Req/Sec	*Request/Second*
RR	*Round Robin*
SSH	*Secure Shell*
Stdev	*Standard Deviation*
URL	*Uniform Resource Locator*
VM	*Virtual Machine*
WLC	*Weight Least Connection*
WRR	*Weight Round Robin*

Chapter One

General Introduction

Chapter One
General Introduction

1.1 Introduction

Load balancing is getting more popular as a novel method for large-scale distributed computing. Many data and computing tasks are now performed in large data centers rather than on desktop and portable computers [1].

One of the most significant aspects a distributed system is load balancing. It's a system that uniformly distributes local workload among all machine nodes to prevent situations where some nodes are heavily loaded while others are idle or doing little activity. It helps to improve overall performance by increasing existing and resource utilization rates. It guarantees that no one node is overworked, enhancing the system's overall performance. Load balancing can allow you to make use of all the resources while saving them. It also improves the implementation of fail-over and scalability, minimizing bottlenecks and decreasing response time [2].

Docker is a virtualization technology that is open source and is recognized as a containerization platform for software containers. These containers provide a method for containing a program, together with the application's own filesystem, inside a single package that is able to be replicated.

Docker is a platform that was developed in 2013 by Solomon Hykes specifically for the Linux operating system. Since its inception, Docker has gained widespread popularity among developers and providers of cloud services due to its capacity to simplify and automate the process of creating and deploying containers, which has contributed to its widespread adoption. Containers are quickly becoming the method of choice for virtualization because container technology enables developers to encapsulate a program along with all of its

dependencies into a standardized unit. This makes containers an attractive option to the traditional method of using VMs. The automation provided by Docker has been essential to the achievement of that goal [3].

A Docker Swarm is a group of machines, either real or virtual, that are running the Docker program and have been configured to join together in a cluster. These machines can be used to run Docker in either standalone or cluster mode. You will still be able to perform the Docker commands that you are accustomed to, but they will now be carried out by the machines that are part of your cluster. This will occur once a set of machines have been clustered together. A swarm manager is responsible for managing the operations of the cluster, and the individual computers that are a part of the cluster are referred to as nodes.

Docker Swarm is a tool for containerized applications developed by Docker. This indicates that the user is granted the ability to exercise control over numerous containers which are executed on separate host computers. The high degree of availability that is provided for applications is seen as one of the most significant advantages connected with the functioning of a docker swarm. Docker swarms generally consist of several worker nodes and at least one manager node. The manager node is the one responsible for managing the resources of the worker nodes in an effective manner and ensuring that the cluster functions in an effective manner [4].

1.2 Thesis Problem

When a lot of people try to use web servers at the same time, it can cause web servers to become overloaded. This is especially true with the growth of data flow on the World Wide Web. The resources are limited due to the use of a single backend web server and therefore cannot handle a large number of requests. In some scenarios, multiple requests cannot be processed in parallel synchronously at the same time. Most websites have a single point of administration (manger),

which is inefficient for centralized IT management. The majority of servers upgraded by vertical scaling (scale-up) have a higher hardware cost.

1.3 Thesis Objectives

The thesis goal is to build a system that uses the main objectives of load balancing to achieve the following objectives:

1. Improve web network traffic performance significantly by increasing the number of requests processed in a given amount of time.
2. Have a fallback strategy ready in case even a portion of the system fails.
3. Keep order throughout in the system, keep the system reliable and stable.
4. Provide future enhancements to the system.
5. Build a pool of services using Docker swarm containers.
6. Then, build a load balancer using Nginx and enhancing an algorithm to distribute the load of network requests to a number of web services using Node JS.
7. After, improving the processing in CPUs Central Processing Unit that use all of their resources to process the requests.
8. Then, use Redis and MongoDB to figure out how to fix any problems that come up as a result of using this system.

1.4 Thesis Impact

This work is helpful within the scope of the application for the National Communications and Media Commission of Iraq (NCMC) in treating load balancing among many instances of an application as a standard strategy for maximizing resource utilization, reducing latency, and ensuring configurations are fault-tolerant.

1.5 Related Works

The following paragraphs explain the most related works associated with load balancers in this section of the thesis:

According to Abdul Aziz and Topan Tampati (2015), the performance of the Apache and Nginx web servers is being compared to find out which one is superior to the two web servers. The methods used are a literature study on web servers, discussion and observation of efforts to determine the most appropriate software to use for web servers, and testing. From the steps taken, it can be concluded that a web server using Apache is superior to a web server using Nginx. Apache's performance in data transfer, connection, and data requests is better, making it easier for clients to get the information they need [5].

According to Poonam Kumari and Mohit Saxena (2016), the Round Robin technique is used to explain how load balancing works and distributes loads. According to the findings, load balancers with multiple servers provide more efficient results in terms of performance and accuracy than single servers. Additionally, the traffic load normally received by a single server is reduced because it is distributed across all servers connected to load balancers [6].

According to Alam Rahmatulloh and Firmansyah MSN (2017), load balancing is a system for evenly distributing traffic loads across two or more connection lines, allowing traffic to flow as efficiently as possible, maximizing throughput, reducing response time, and avoiding overloading one connection line. Load balancing is utilized when a server's maximum capacity is reached. To enhance resource usage, load balancing spreads workloads evenly among two or more computers, network lines, CPUs, hard drives, or other resources [7].

According to Intan Ferina Irza, Zulhendra, and Efrizon (2017), Apache and Nginx are two web servers that are commonly used nowadays. Users expect a media content supplier to meet all of their expectations, particularly in terms of

device performance. The authors wish to examine and compare the performance of both the Apache and Nginx Web servers so that users can pick the best one. By allocating load to each test and performing on attributes that exist, the author merely compares the parameters of throughput, connection, request, reply, and error. After testing, the results revealed that NGINX outperformed Apache in terms of replying to and connecting to data required by the client [8].

According to Ruoyu Li, Yunchun Li, and Wei Li (2018), they analyze the web application deployment architecture using Docker containers, generate host load metrics based on the host's performance indicators and operational container status, and suggest a dynamic weighted least connection technique. In comparison to the round-robin method and the least-connection algorithm, the results prove that this approach can effectively increase the web cluster's response speed [9]. The difference between this work and our proposal is that the method used does not assign a weight to each server. Instead, the weight is assigned to all servers as a single value based on the total response number. This flaw is a waste of resources since it does not provide an accurate or estimated weight for each server. Additionally, PHP is recommended as a back-end language (Single-threaded).

According to Gurasis Singh, Kamalpreet Kaur (2018), this work suggested an improved WLC (Weight Least Connection) algorithm that maintains load balance across real servers by preventing web requests from being allocated to only a new real server when it is added to an active real server list for the first time. Web cluster systems often employ the WLC load balancing algorithm, which assigns a throughput weight to actual servers and selects the least connected server to execute web requests. When a new real server with multiple simultaneous clients is introduced to a web cluster system, the previous WLC scheduling mechanism distributes web requests across the many real servers, causing a load imbalance. The suggested technique is made up of two parts: a real

server selection mechanism and an overload reduction algorithm. A real server is chosen using the proposed method's real server selection algorithm, and the chosen real server is supplied to the overload decrease algorithm as a parameter. If a web request is assigned to a real server more than C times, the real server is set to a deactivated state and excluded from the activated real server list in the overload reduce algorithm, and the new real server is added to the real server scheduling list by activating it after C-1 allocation round times. As a consequence, by preventing overloads on the new real server, the proposed technique maintains load balance across actual servers [10]. The difference between this work and our proposal is that the method used does not assign only overhead dependent. Rather, it is assigned to each server as a number value based on the C. This represents the maximum number of requests that have been routed to the server and makes this server inactive for a time. This flaw is a waste of resources since it does not provide an accurate or estimated weight for each server by using the number of C times as the maximum connection to make the server inactive for some time. If the load balancer receives a large number of requests, drop requests become impossible to route requests to any server. Additionally, this work does not use a back-end language to process requests in CPU but instead uses a website of the HAProxy service to check tests.

According to Dimara Kusuma Hakim, Dwi Yoga Yulianto, and Achmad Fauzan (2019), the server delivering the request needs to work harder when the volume of traffic grows. Server performance decreases as a result, and these online services encounter frequent outages. The village service information system for the Purworejo district is a web-based application that saves and administers village information and services. The lack of consistent connection speed is one of the issues clients have while working with this website. Because so many users use Ubuntu, Oracle Virtualization, and Nginx load balancing at the same time, web server loads may be shared when requests are large, reducing web

server traffic pressure. When a significant number of user requests arrive at the same time, In terms of website access performance, the outcomes of employing the least connection strategy for load balancing surpass those of the round-robin strategy [11].

According to Luthfan Hadi Pramono, Robby Cokro Buwono, and Yanuar Galih Waskito (2019), in this study, a load balancing performance review will be carried out on HAProxy and Nginx by implementing the Round-robin algorithm on both load balancing servers. Load balancing makes web application performance better when more than one server is used. This can improve the efficiency of the results created by the server and reduce the traffic load from a single server by distributing the traffic load to all servers connected to the load balancer. The results of the final conclusions of this study are that a load balancing server has better performance, which can later be used as a descriptive and measurable reference for future studies [12].

According to Kresna Adi Pratama, Ridho Taufiq Subagio, Muhammad Hatta, Victor Asih (2021), the load balancing technique uses a request counting algorithm to distribute the load equally on the web server and reduce client-server response time. The load is distributed across server members who have registered with the load balancing server. Because of the high availability system, the work of the server will be maximized when using the load balancing technique, because when one server fails, the work of the server will be taken up by another server. A system with a mirror server, in addition to the load balance technique, may assist in optimizing the load balance method by automatically replicating website content and databases across web servers that are members of the load balance. As a consequence, the company's web server will become a system capable of providing excellent web server services to customers since the load will be evenly distributed and the response time will be short, ensuring that clients will have no problem accessing the company's website [13].

According to Murugesan, P, Prakash, V S, Srisabarishkannan, Mr V P (2021), the major goal of this study is to lower the document distribution load and client request load on the web server in a DWS (Distributed Web Server) system. The goal of this study was to find a way to deal with web server failures. The rules employed in a document sent using a DWS system are examined in this study. To address the loss of the web server, this research paper provides a different web server idea. Based on the contents it seems to be a mirror server. In addition, the alternative server watches all of the primary web server's operations and takes the appropriate actions to address any web server failures. A distinct load balancing model is used in this system, which proposes a partial document distribution architecture. In this system, the schema is used. The load is assessed using this schema, which takes into account the number of requests and the estimated response time. All of the submitted ideas are put into action, and their results are evaluated. During the run-time of the DWS system, the heuristics were able to balance the load among the server nodes, and the traffic caused by document transfer was minimal [14].

Table 1.1 A summary of related works

Reference	Aims of the Work	Algorithm	Method	Year
[5]	This work aims to provide an analysis of the performance of the two web servers with the parameters of transfer rate, time per request, and connection time as measurement indicators for testing the performance of the two web servers. The conclusion of the analysis from the test results is that the Apache web server has better performance than the Nginx web server.	Round Robin	Apache, Nginx	2015
[6]	In this research, a load balancer was developed based on the Round Robin Approach. A load balancer receives hundreds of requests at the same instant from a client and it distributes the load to the different instances/machines. Load distribution is independent of the number of instances/machines. This can minimize the response time of a program.	Round Robin	Nginx, PHP As Backend	2016
[7]	This work performs file synchronization well as files uploaded to node 1 will sync	Round Robin	HAProxy, PHP As Backend	2017

	to every node 2 and node 3 in the cluster. Implementing server aggregation can increase the response time of the web server and increase the number of existing connections the server can serve.			
[8]	The authors want to analyze and compare the performance of both Web servers are Apache and Nginx, so users can choose the best Web Server. The author only compares the parameters of throughput, connection, request, reply and error by assigning load to each test. After testing, the results obtained where in responding and connecting data that requested by the client of web application server Nginx was superior to apache.	Round Robin	Apache, Nginx	2017
[9]	This paper examines the container-based WEB application deployment architecture and calculates host load metrics, as well as the host's performance indicators and the status of the operating containers, such as CPU utilization, memory utilization, network utilization, and the proportion of unused memory allocated to the containers.	Round Robin, Least Connection, Dynamic Weight Least Connection (DWLC)	Nginx, PHP As Backend	2018
[10]	The proposed improvement in this work is an improved WLC algorithm that maintains load balance among real servers by avoiding web requests depending on the selected real server being passed to an overload reducer algorithm as a parameter and web requests are assigned to a real server no more than C times.	Weight Least Connection, Improved Weight Least Connection (IWLC)	Nginx, HAProxy As Backend	2018
[11]	This work aims to increase access speed when there are many requests from users at the same time on a website-based application that functions to store and manage information and services.	Round Robin, Least Connection	Nginx	2019
[12]	Evaluations of HAProxy and Nginx servers, if the Keep Alive configuration on Nginx is activated, then Nginx has very good performance and faster than the HAProxy.	Round Robin, Least Connection	HAProxy, Nginx	2019
[13]	The server load divided and will be served according to the web server route, maximizing the availability of web server services, establishment of a failover system.	Round Robin	Apache	2021
[14]	The major goal of this study is to lower the document distribution load and client request load on the web server in a distributed web system and find a way to deal with web server failures. This schema calculates the load based on the number of requests and the estimated response time.	Distributed Web Server (DWS)	Partial Document Distribution Architecture	2021

1.6 Outlines of Thesis

In addition to chapter one, there are four additional chapters in this thesis:

Chapter Two introduces Cloud Computing, Virtualization, Docker, Docker Swarm, Load Balancing, Nginx, Node JS, Express JS, APIs, Redis, and MongoDB.

Chapter Three introduces the proposed system, depicts the system's practical stages, and explains the system.

Chapter Four has presented the results and evaluated the proposed system (proposal algorithm in Nginx and Node JS as single-threaded and multi-threaded).

Chapter Five presents the conclusion, thesis limitations, and it also includes suggestions or offers ideas for future works.

Chapter Two

Theoretical Background

Chapter Two
Theoretical Background

2.1 Introduction

This chapter provides a background about cloud computing, Virtualization, Docker, Docker Swarm and technology that use of load balance in network. Moreover, some of the methods that can be used to distribute the traffic with the best accuracy and performance of resources.

2.2 Cloud Computing

Cloud computing refers to the use of the internet to deliver computing services like servers, storage, databases, networks, software, analysis, and intelligence in order to provide more flexible and speedier innovation resources as well. There are advantages to having a lot of things in one place. Most of the time, you only pay for the cloud services that you use, so you don't have to pay for anything else. This helps you reduce operational expenses, manage the resources of your infrastructure more effectively, and scale your organization according to your needs. Cloud computing is a type of distributed system, and therefore, it requires efficient distribution of loads between servers using a load balancer [1].

2.3 Virtualization

Virtualization is the method of creating virtual representations of objects or resources such as servers, storage devices, networks, and operating systems. A machine running multiple operating systems at the same time is referred to as virtualization. Virtual machines allow applications to run as if they were on their own dedicated machine, with their own operating system, libraries, and other programs that are not related to the main host operating system [1] [3].

2.4 Docker

Docker is an open-source application for automating application deployment into containers. Docker is an open-source platform based on container technology. A container image is a lightweight independent software package that contains everything needed to operate it, including code, runtime, system tools, system libraries, and settings. Containers make use of the components required to run the software. Files, environment variables, dependencies, and libraries are examples of these components. As shown in figure 2.1, containers differ from virtualization in that they have a smaller file size because they don't require the installation of an entire operating system [15].

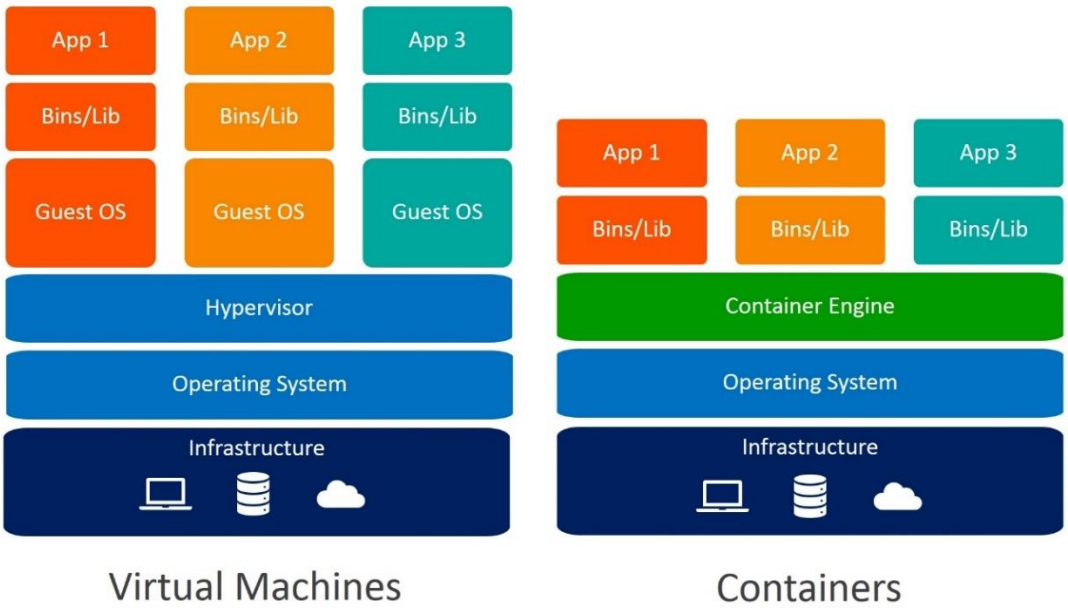


Figure 2.1 Difference between a Virtual Machine and a Docker Container

Containers are a software development solution offered by Docker. According to Docker, a container is “a lightweight, independent, executable package of software that includes everything needed to run it”. Since containers are platform-independent Docker can run on both Windows and Linux-based platforms. If necessary, Docker can also be operated inside a virtual computer. Docker's main function is to allow you to run microservice applications in a distributed architecture [16].

Consider this: Docker is the best since it is lightweight, cost-effective, and scalable. Docker is a container technology that allows users to create distributed applications [17].

2.4.1 Docker Architecture

Docker has a client-server architecture. However, given the features required, it's a little more difficult than a virtual machine. As shown in figure 2.2, it is divided into four sections [18]:

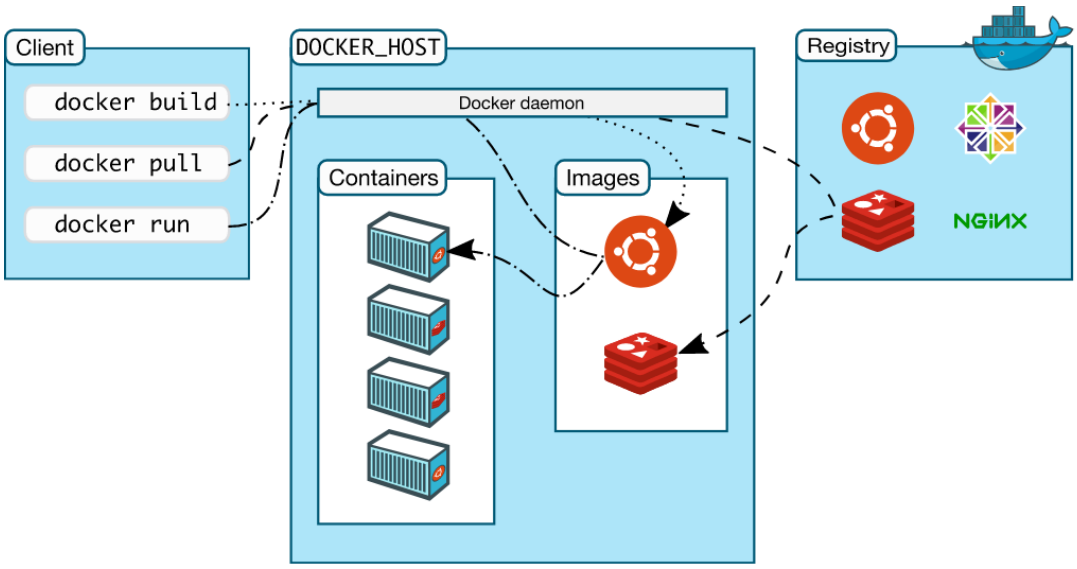


Figure 2.2 Docker Architecture [18]

1. Docker Client: This is the application that allows you to interface with your containers. It's referred to as Docker's user interface.
2. Docker Objects: Your containers and images are the fundamental components of Docker. Containers, as previously stated, are placeholders for your software that may be read and written to. Container images are read-only files that are used to build new containers.
3. Docker Daemon: A background process that receives commands from the command line and passes them on to the containers.
4. Docker Registry: Also known as Docker Hub, this is where you store and get your container images.

2.4.2 Real-World Use Case For Docker

PayPal infrastructure use Docker to improve "cost efficiency and enterprise-grade security." PayPal believes that running virtual machines (VMs) and containers side by side reduces the number of VMs it needs to run [19]. Ways, for example, to use Docker for developing apps:

- Application development: Docker is generally used to package the code and dependencies of an application. The development pipeline can be portable by sharing the same container from development (Dev) to quality assurance (QA) and then to information technology (IT).
- Using Docker to execute microservices applications: Docker allows you to run each of the microservices that make up an application in its own container. It allows for a distributed architecture in this fashion.

2.4.3 Advantages Of Docker Containers

Docker is growing in popularity among large organizations. It's obvious that speed and efficiency are the most important requirements for DevOps teams, and Docker outperforms VMs at this point. Docker's ability is being recognized, even if it hasn't totally replaced virtual machines in production scenarios. This is not to say that virtual machines won't be gone in the near future. Rather than Docker, virtual machines will coexist, offering DevOps teams greater options for running cloud-native apps [19].

1. Docker containers are process-isolated and do not require the use of a software hypervisor. Docker containers are therefore much smaller and use far fewer resources than virtual machines.
2. Docker is a lightning-fast platform. While it can take several minutes for a virtual machine (VM) to boot and become development-ready, starting a Docker container from a container image can take anything from a few milliseconds to (at most) a few seconds.

3. Containers may be shared by various members of a team, providing much-needed flexibility throughout the development process. This helps DevOps (Developers & Operations) teams avoid problems like "works on my computer".

2.4.4 Docker Properties

1. Docker is a tool that can assist both developers and system administrators, it's included in many Dev Ops technologies that are available.
2. For developers, this means they can be focused on developing code rather than worrying about what operating system it will run on.
3. It also enables users to get a start by adding one of the hundreds of programs built to operate in a Docker container into their apps.
4. Docker provides flexibility to operations workers and minimizes the number of systems required because of its small size and minimal overhead.

2.5 Docker Swarm

Docker Swarm is a docker-native clustering solution that allows numerous Docker hosts to be combined into a single large virtual server. A swarm is made up of many docker servers, all of which are running in cluster swarm mode, with some functioning "as managers (to administer membership) and as workers (to perform swarm services)". One of the key benefits of the swarm service is that you can change its configuration, including the network and volumes connected to it, without having to restart it manually, including the networks and volumes to which it is linked. Docker will refresh the configuration, terminate all service jobs of our configurations, and restart them with the new configuration [4] [20].

A swarm kit is used to build the Docker Engine cluster management capabilities. The Swarm kit is a self-contained project that provides the Docker orchestration layer.

Many Docker servers may operate as supervisors. When utilizing swarm mode, you may manage members and authority, as well as workers in a swarm. (which runs swarm services). A Docker node may be both a supervisor and a worker at the same time. When you start a service, you write down what its ideal state should be. This includes how many replicas, network, and storage resources are available, how many ports the service allows the outside world to see, and other things you think are important. Docker makes every effort to keep the desired state as much as possible. If a worker node becomes unavailable, Docker, for example, distributes its tasks to other nodes. A task, as opposed to a solo container, is a running container that is handled by a swarm manager as part of a swarm service.

While Docker is in swarm mode, you may continue to execute independent containers and swarm services on any of the swarm's Docker servers. Solo containers differ from swarm services in that swarms can only be managed by swarm managers, but solo containers can operate on any server. In a swarm, Docker modules can operate as supervisors, workers, or both. Swarm service stacks are built and run in the same way that containers are built and run with Docker Compose [20].

Continue reading for more information about Docker swarm concepts such as nodes, services, and tasks.

2.5.1 Nodes Of Docker Swarm

A node is a Docker engine instance that is part of the swarm. This can also be considered a Docker node. On a single physical machine or cloud server, one or more nodes can execute. Production swarm deployments, on the other hand, often include Docker nodes scattered over several physical and cloud servers.

You send a service definition to a management node to deploy your application to a swarm. The management node assigns tasks to worker nodes, which are units of work.

The coordination and cluster management operations necessary to keep the swarm in good conditions are also performed by manager nodes.

Worker nodes get tasks from management nodes and perform them. Management nodes use worker nodes to run services by default, but they can be configured to do just manager responsibilities and act as manager-only nodes. Each worker node has its own agent, which performs and reports on the tasks that have been allocated to it. The worker node tells the management node about the current status of its job responsibilities, allowing the manager to verify that each worker is fully operational [20].

2.5.2 Tasks And Services

Managers or workers nodes are given tasks by a service that tells them what they need to do. It is the swarm system's core structure and the principal point of user interaction with the swarm. When you build a service when you build a service you specify which container image to use and which commands to execute inside running containers. A set number of replica tasks is distributed across the nodes by the swarm controller in the replicated services model based on the scale you set in the intended state.

On each available node in the cluster, the swarm does one job for each global service. A Docker container and the instructions to run it make up a job. It is the swarm's primary scheduling unit. Based on the number of replicas defined in the service scale, manager nodes allocate workloads to worker nodes. A task given to one node cannot be moved to another. This can run only on the node to which it was assigned. Otherwise, it will be doomed to fail [20].

2.5.3 Function Of Nodes

Swarm mode is a new feature of the Docker Engine. You may use this to create a swarm within one or maybe more Docker Engines. A swarm consists of one or even more nodes, which may be real or virtual computers that are configured to execute Docker Engine in swarm mode. Managers and workers are the two kinds of nodes [20].

A. Nodes That Serve As Managers

Cluster management responsibilities are handled by manager nodes:

- Keeping the cluster state up to date
- Scheduling services
- Providing HTTP API swarm mode endpoints

As shown in figure 2.3, managers may keep track of the internal status of the entire swarm, including all services operating on it, using a Raft implementation (Raft consensus). Raft provides a general technique to distribute a state machine over a cluster of computers, ensuring that each node agrees on the same set of state transitions. For testing reasons it's fine to control a swarm with a single manager. Your services will continue to operate if a single-manager swarm's manager fails but you'll have to start anew with a new cluster.

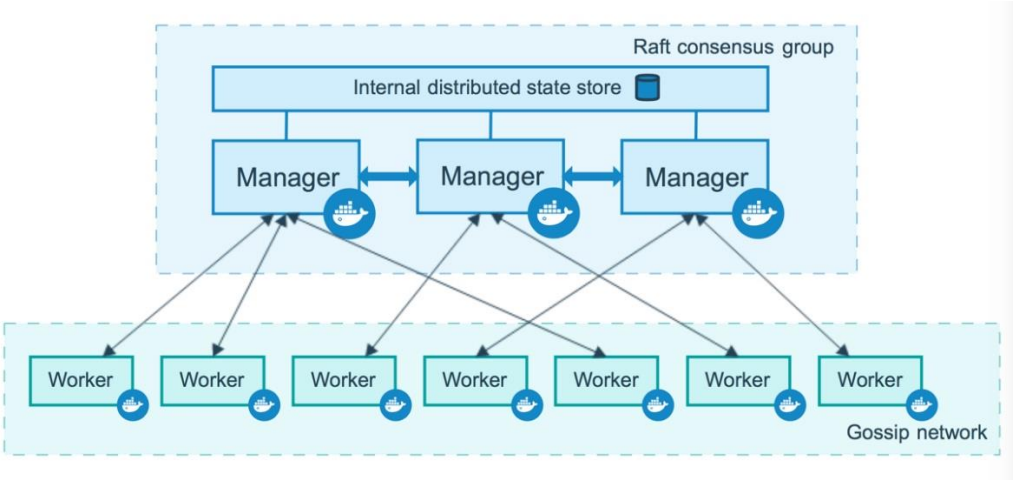


Figure 2.3 Raft consensus in swarm mode [20]

Depending on your organization's high availability needs, Docker recommends utilizing an odd number of nodes to take use of swarm mode fault-tolerance features. You can quickly recover from a management node failure when you have a high number of managers.

- At any one moment only one of the three managers in a three-manager swarm may die.
- A five-manager swarm may survive the loss of up to two management nodes at the same time.
- A cluster of N managers can endure the loss of up to (N-1)/2 managers.
- According to Docker a swarm should have a maximum of seven management nodes.

B. Nodes With Workers

Worker nodes are Docker Engine instances dedicated to container execution. Worker nodes do not participate in the Raft distributed state, do not make routing decisions, and do not support the swarm mode HTTP API.

One management node may generate a swarm, but no worker node can exist without at least one manager node. All managers are also employees by default. You may execute commands as the Docker service generates them in a single management node cluster, and the scheduler will perform all tasks on the local engine.

Set the management node's availability to Drain to prevent the scheduler from putting tasks on it in a multi-node swarm. Tasks on nodes in Drain mode are gently stopped and rescheduled on active node by the scheduler. No additional work is allocated to nodes with Drain availability by the scheduler.

C. Switch Of Roles Node

Docker node promote may be used to promote a worker node to a management node. When you take a management node down for maintenance,

for example, you may wish to promote a worker node. A management node may also be demoted to a worker node.

2.6 Load Balancing

Load balancing is a method of fairly dividing workloads, network traffic, and application traffic over several servers. Load balancing improves the functionality of your system by enabling increased reliability and performance by eliminating the occurrence of a single point of failure. The Round Robin and Least Connections algorithms are two of the most frequently utilized algorithms. Each incoming request is routed to the next server on the list using the round-robin scheduling method. Seeing as the least connection algorithm requires dynamically calculating direct connections for each server, it is one of the algorithms that can be used to make sure things get done quickly [2] [21] [22].

2.7 Algorithms Of Load Balancing

This section shows the algorithms of load balance and explains them in detail.

2.7.1 Round Robin (RR)

Round-robin load balancing is one of the most fundamental and often used load balancing algorithms. Client queries are dispersed between application servers on a rotational basis [23]. The flowchart of this algorithm is shown in figure 2.4.

Algorithm 2.1 below represents the Round Robin (RR) algorithm in a program with code in C++. To represent how a request is routed to the next server.

Algorithm 2.1 : Round Robin (RR)

```
#include <stdio.h> // Library
#include<conio.h> // Library
#include <iostream> // Library
#include <string> // Library
using namespace std; // Library

int requests = 10; // Number Of Requests Come From Clients
const int node = 2; // Number Of Nodes
#define MAX 50  // Maximum Size Of Queue
int intArray[MAX]; // Maximum Size Of Queue
int front = 0; // Index For First Queue
int rear = -1; // Index For Last Queue
int itemCount = 0; // Length Of Elements In Queue
int a = 0; // Begin At The Top Of The List With The First Node

// Check If Queue Full
	bool isFull() {
	return itemCount == MAX;
}

// Check If Queue Empty Or Not Return (1 = True) (0 = False )
bool isEmpty() {
	return itemCount == 0;
}

// Remove One Item From Queue
int pop() {
	int data = intArray[front++];
	if(front == MAX){
		front = 0;
	}
	itemCount--;
	return data;
}

// Insert New Element In Queue
void push(int data) {
	if(!isFull()) {
		if(rear == MAX-1) {
			rear = -1;
		}
		intArray[++rear] = data;
		cout<<data<< " Pushed into Queue\n";
		itemCount++;
	}else{
		cout<<"Queue Overflow\n";
	}
}

int main() {

	// Specify Size Of The Array Nodes
	string matrix[node];

	// Get Size Of Node Array
	#define my_sizeof(type) ((char *)(&type+1)-(char*)(&type))
	int size = my_sizeof(matrix)/my_sizeof(matrix[0]);

	// Input Nodes In List Of Array
	for(int i=0;i<node;i++){
```

```
61                  string text_string = to_string(i+1);
62                  matrix [i] = "Node" + text_string;
63          }
64          // Request Pushed Into Queue
65          for(int k=1;k<=requests;k++){
66                  push(k);
67          }
68
69          // Routed Request N To Node N
70          for (int i=0;i<requests+1;i++){
71                  // Check Queue Is Not Empty From Requests
72                  if(!isEmpty()){
73                          cout<<"Request "<<pop()<<"  ";
74                          cout << matrix [a]<<"\n";
75                  }
76                  // Go To Next Node In List (Matrix)
77                  a=a+1;
78                  // If Arrived End Of List Go To The Begin List
79                  if(a>=size){
80                          a=0;
81                  }
82          }
83  }
```

To show this algorithm's simple design, you can go to figure 2.4 to see the flowchart of the Round Robin Algorithm (RR).

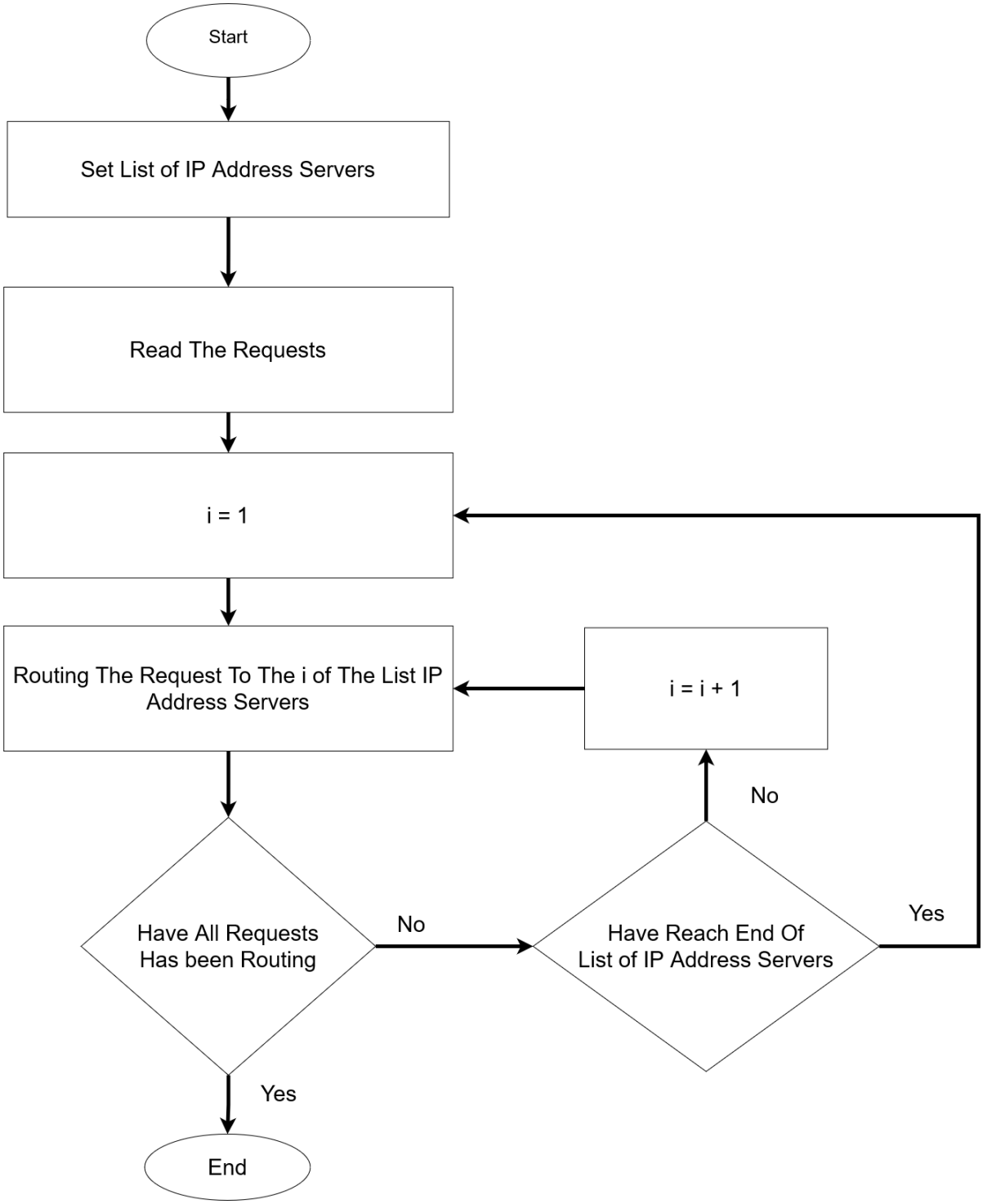


Figure 2.4 Flowchart of Round Robin Algorithm (RR)

As shown in figure 2.5, if you have two servers, for example, the first client request will be sent to the first application server, the second to the second application server, the third to the third application server, and so on.

This load balancing method assumes that all application servers have the same availability, computing, and load-handling capabilities [24].

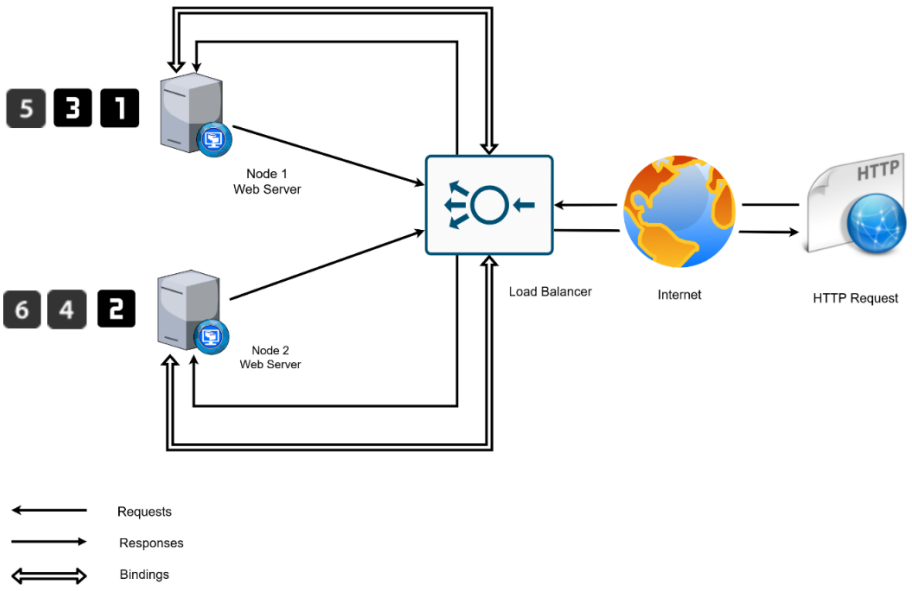


Figure 2.5 Example Work of Round Robin Algorithm (RR)

2.7.2 Least Connection (LC)

Client requests are sent to the application server with the fewest active connections at the moment the request is received while using the least connection load balancing approach. When application servers have similar characteristics and a server is overloaded owing to longer-lasting connections, this technique assesses active connection load [23]. The flowchart of this algorithm is shown in figure 2.6.

Even though the specifications of two servers in a cluster are equal, one server may get overloaded faster than the other. Clients connecting to Server 2 are active and stay connected for substantially longer than clients connecting to Server 1.

In this case, the least connection technique might be a better fit. This approach takes into consideration the number of active connections on each server. When a client attempts to connect, the load balancer searches for the server with the fewest connections and allocates the new connection to it. This algorithm distributes traffic between web servers based on the current connection count between the load balancer and the web server.

Algorithm 2.2 below represents the Least Connection (LC) algorithm in a program with code in C++. To represent how a request is routed to the next server depending on the current connection of the server.

Algorithm 2.2 : Least Connection (LC)

```
1  #include <stdio.h> // Library
2  #include<conio.h> // Library
3  #include <iostream> // Library
4  #include <string> // Library
5  using namespace std; // Library
6
7  int requests = 15; // Number Of Requests Come From Clients
8  const int node = 4; // Number Of Nodes
9  #define MAX 50  // Maximum Size Of Queue
10 int intArray[MAX]; // Maximum Size Of Queue
11 int front = 0; // Index For First Queue
12 int rear = -1; // Index For Last Queue
13 int itemCount = 0; // Length Of Elements In Queue
14 int a = 0; // Begin At The Top Of The List With The First Node
15
16 // Check If Queue Full
17        bool isFull() {
18        return itemCount == MAX;
19 }
20
21 // Check If Queue Empty Or Not Return (1 = True) (0 = False )
22 bool isEmpty() {
23        return itemCount == 0;
24 }
25
26 // Remove One Item From Queue
27 int pop() {
28        int data = intArray[front++];
29        if(front == MAX){
30               front = 0;
31        }
32        itemCount--;
33        return data;
34 }
35
36 // Insert New Element In Queue
37 void push(int data) {
38        if(!isFull()) {
39               if(rear == MAX-1) {
```

```
40                      rear = -1;
41              }
42              intArray[++rear] = data;
43              cout<<data<< " Pushed into Queue\n";
44              itemCount++;
45      }else{
46              cout<<"Queue Overflow\n";
47      }
48  }
49
50  int main() {
51
52      // Specify Size Of The Array Nodes
53      std::string matrix [node][2];
54
55      // Get Size Of Node Array
56      #define my_sizeof(type) ((char *)(&type+1)-(char*)(&type))
57      int size = my_sizeof(matrix)/my_sizeof(matrix[0]);
58
59      // Input Nodes In List Of Array
60      for(int i=0;i<node;i++){
61              string text_string = to_string(i+1);
62              // [server node]     [Current Connection]
63              matrix [i] [0] = "Node" + text_string;    matrix [i] [1] = "0";
64      }
65
66      // Request Pushed Into Queue
67      for(int k=1;k<=requests;k++){
68              push(k);
69      }
70
71      // Routed Request N To Node N
72      for(int j=1;j<=requests;j++){
73              /* "min" Is Value That Storage The Lowest Number
74               Of Current Connection Of Nodes */
75              int min = atoi(matrix[0][1].c_str());
76              // Pointer To Select The Node With The Lowest Minimum
77              int i_matrix_in_number=0;
78              /* All Nodes Must Be Tested In Order To Achieve A Low Minimum
79               Current Connection And Storage Value */
80              for(int i=0;i<size;i++){
81                      if(atoi(matrix[i][1].c_str()) < min){
82                              min = atoi(matrix[i][1].c_str());
83                              i_matrix_in_number=i;
84                      }
85              }
86
87              /* The Request Is Routed To The Node That Has
88               The Minimum Current Connection */
90              cout<<"Request: "<<pop()<<" Current Connection: "<<min
91                      <<" Node: "<<matrix[i_matrix_in_number][0]<<"\n";
92
93              /* Then, After The Node Has Selected And Routed The Request,
94               Make The Current Connection Of This Node Increase By One */
95              int s = atoi(matrix[i_matrix_in_number][1].c_str())+1;
96              std::string ss = std::to_string(s);
97              matrix[i_matrix_in_number][1]=ss;
98      }
99  }
```

To show this algorithm's simple design, can go to figure 2.6 to see the flowchart of the Least Connection Algorithm (LC).

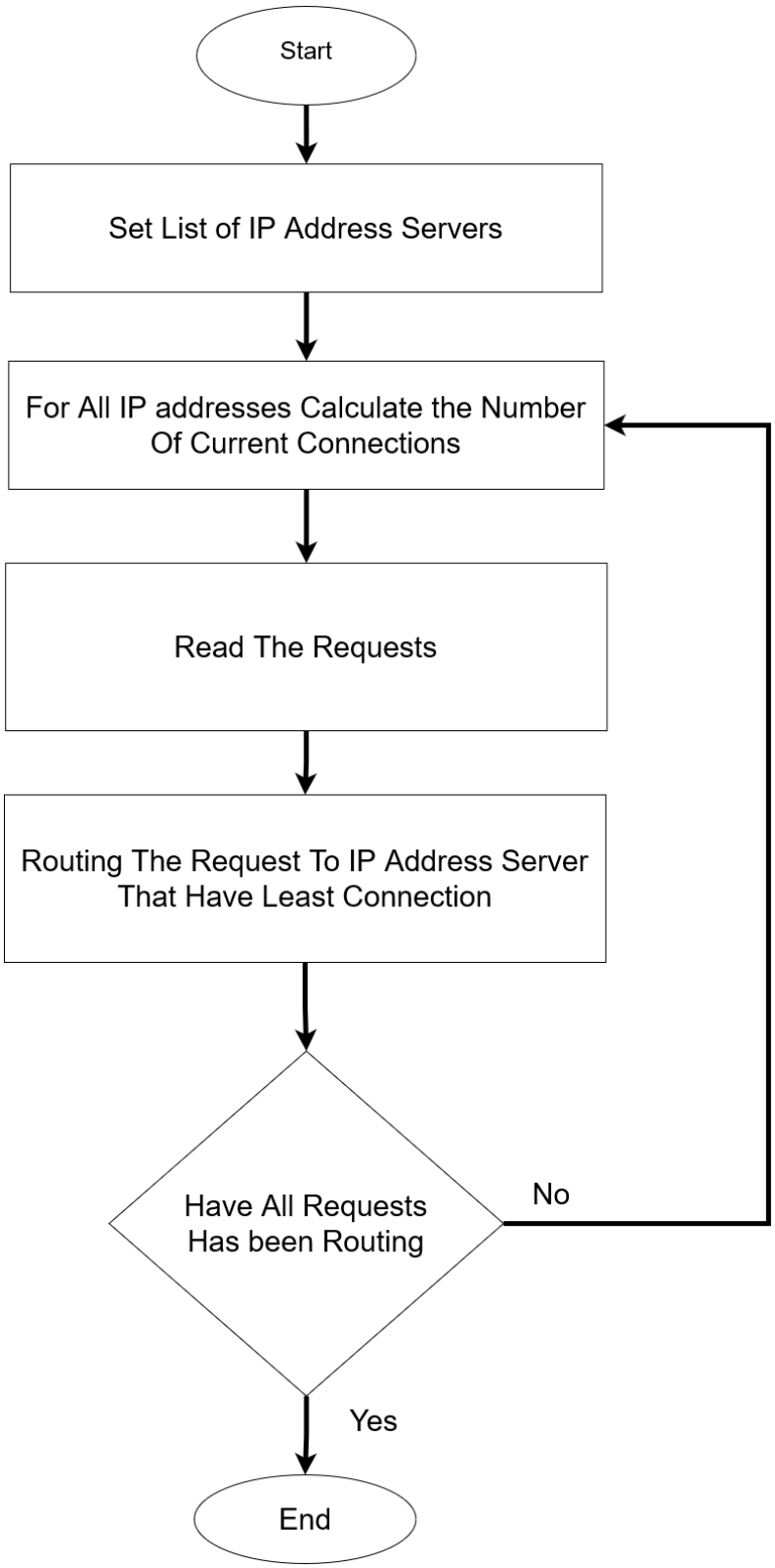


Figure 2.6 Flowchart of Least Connection Algorithm (LC)

2.8 Nginx

NGINX (pronounced "engine X") is a reverse proxy that may be used as a load balancer, a mail proxy, or an HTTP cache. Nginx is a free-to-use open-source web server. A large number of web servers utilize NGINX as a load balancer [25] [26].

“Igor Sysoev” built the software, which was publicly published in 2004. In 2011, a company with the same name was created to provide support and premium software called Nginx Plus. F5 Networks bought the firm for $670 million in March 2019.

NGINX is one of a few servers created to address the C10K issue. The C10k issue occurs when network sockets are tuned to serve a large number of simultaneous consumers.

2.8.1 Nginx Work

Instead of creating separate processes for each web request, Nginx uses an asynchronous, event-driven architecture that handles requests in a single thread.

Nginx enables a single master process to oversee several worker processes. The master oversees the worker procedures while the workers carry out the actual job. Nginx is asynchronous; therefore, the worker may handle several requests at the same time without causing other requests to be stopped.

It was developed to handle thousands of simultaneous HTTP requests and connections and to serve content efficiently and quickly. Because of NGINX's event-driven design, which avoids the need to create a new process each time a request is received, CPU and RAM usage should be more consistent. NGINX grows in all directions, from tiny virtual private servers to huge clusters of servers. Netflix, CloudFlare, GitHub, SoundCloud and Heroku, among other great websites, run on NGINX.

Nginx has a number of common features, including:

1. NGINX is free, open-source software.
2. It offers a configuration model based on JSON.
3. Exposes that configuration via an HTTP API.
4. It provides a declarative model for configuring the most common networking use cases.
5. It is lightweight and simple to run.
6. Low resource consumption.
7. Simple configuration
8. High performance and stability.
9. Static file management, index file management, and auto-indexing.
10. Enable TLS/SSL.
11. Load balancing.
12. A reverse proxy that caches data
13. Support IPv6.

The capability of NGINX currently meets business needs, thus it's the clear option over establishing the web service [27].

2.9 Node JS

The most recent version of Node JS is 10.18.1, which was created by Ryan Dahl in 2009. The definition below comes from the official NodeJS documentation. The JavaScript Engine in Google Chrome powers the Node JS server-side platform (V8 Engine) [28] [29].

Node JS is a framework for building fast and scalable network applications that are built on Chrome's JavaScript engine. It's perfect for real-time content applications that span several platforms. Node JS is lightweight and efficient because of its event-driven, non-blocking I/O design.

Node JS is a cross-platform runtime environment for developing server-side and networking applications that is open-source. On Windows and Linux, Node JS apps are developed using JavaScript and run on the Node JS runtime. Node JS also includes a large variety of JavaScript modules that make developing Node JS web applications a lot simpler.

Runtime Environment + JavaScript Library = Node JS

Node JS is used by organizations such as GoDaddy, IBM, LinkedIn, Microsoft, Netflix, PayPal, Yahoo, and Amazon Web Services.

2.9.1 Features Of Node JS

There are a lot of features in Node JS. The following are some of the important aspects that make Node JS the programming language of choice for software engineers.

1. Asynchronous and Event-Driven: The Node JS library's APIs are totally asynchronous, which means they don't block. In the simplest sense, it is a node. A JavaScript-based server never has to wait for data from the API. After visiting an API, the server moves on to the next API and the Node Events notification mechanism. With the aid of JavaScript, the server obtains the result of the previous API request.
2. Very Fast: Because it is based on Google Chrome's V8 JavaScript Engine, the Node JS framework is highly quick in terms of code execution.
3. Single-Threaded but Scalable: Because Node JS employs a single-threaded technique with event looping, it is scalable. Unlike conventional servers, which produce a limited number of threads to process requests, the event mechanism allows the server to reply non-blocking, allowing it to be more scalable. Node JS is a single-threaded

application capable of handling more requests than standard HTTP servers such as Apache HTTP Server [30].

4. No data buffering: Data in Node JS apps is never buffered. These programs merely generate the data in segments.

2.9.2 Node JS Non-Blocking

Asynchronous (async) execution is when code is executed in a different order than it appears in the code. The program in async programming does not have to wait for the job to finish before moving on to the next one.

Non-blocking I/O operations enable a single process to handle numerous requests concurrently. Instead of blocking the process and waiting for I/O operations to complete, the system delegate the I/O operations to the process, allowing the process to go on to the next piece of code. When a non-blocking I/O operation is completed, a callback function is called [28].

2.9.3 Who Makes Use Of Node JS

A comprehensive collection of projects, apps, and enterprises built using Node JS. Just a few of the firms on this list include eBay, GE, GoDaddy, Microsoft, PayPal, Uber, and Yahoo.

2.9.4 When Should Node JS Be Used

The following are some of the areas where Node JS has been shown to be a good technology partner.

- Data Streaming Applications.
- Applications that need a lot of data.
- Applications that depend on I/O.
- The term "real-time" refers to apps that run in real-time (DIRT).
- Applications based on JSON APIs.
- Applications that are only one page long.

2.10 Redis

Redis is an open-source data structure storage software that may be used as a distributed, in-memory key-value database, cache, and message broker. It also comes with a strength choice. Redis supports a variety of abstract data structures, including strings, lists, maps, sets, sorted sets, bitmaps, streams, and spatial indexes [31] [32] [33].

2.11 MongoDB

MongoDB is a popular NoSQL database that is also available as an open-source document database. MongoDB is written in C++. MongoDB is a scalable and performance-oriented database [34] [35] [36] [37].

MongoDB is a document-oriented database that runs on a variety of systems and offers high availability, great performance, and easy scalability. MongoDB is built on the notions of collections and documents [38] [39].

- The database is used to store the collections physically.
- Collection: A collection of MongoDB documents is referred to as a "collection." In a relational database management system, it's the same as a table (RDBMS). Within a single database, a collection may exist. A schema is unaffected by collections. Many fields may be found in different documents within a collection. In most cases, the pieces in a collection are all directed at the same or comparable objectives.
- Document: A collection of key-value pairs that make up a document is referred to as a "document." The schema of documents is always evolving. A collection's documents do not all have the same set of fields or structure, and common fields in a collection's documents might include a wide range of information.

What Makes MongoDB So Special?

- Document Oriented Storage: Data is saved in JSON format.
- Any attribute may be indexed on
- High availability and replication
- Auto-sharding
- Rich queries
- In-place modifications that happen quickly

When Is MongoDB Appropriate?

- User Data Management
- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure

2.12 Express JS

Express JS is a Node JS framework that enables programmers to create and maintain dependable servers. It comes with a lot of built-in features and a lot of third-party add-ons that developers may use to increase functionality, security, and speed. It is designed for the development of web applications and APIs [40].

2.13 Application Programming Interface

An application programming interface (API) is a collection of rules that specify how several software programs or hybrid hardware-software agents communicate. It describes, the sorts of calls or requests that may be made, how they should be made, the data formats to be used, and the protocols to be followed. It can also include extension features, which allow users to customize and extend existing functionality in a number of ways. An API might be totally unique, customized for a component, or based on an industry-standard to assure

compatibility. APIs assist modular development by hiding data, allowing consumers to use the interface independent of implementation.

APIs are extensively used programming interfaces. They let the program connect with other software both inside and outside, which is necessary for scalability and dependability. Online services that expose APIs to the public are becoming increasingly common these days. Other app developers may use them to quickly add features like social networking logins, credit card payments, and behavior monitoring to their applications. REST (Representational State Transfer) is the accepted standard for all of this [41].

2.14 REST

REST is a collection of structural constraints rather than a protocol or standard. REST may be used by API developers in a variety of ways. A RESTful API responds to a client request by returning a representation of the resource's state to the requester or endpoint. HTTP delivers this data in a number of forms, including JSON (JavaScript Object Notation), HTML, and plain text. Because it is language neutral and can be used by both people and computers, JSON is the most widely used programming language.

A REST API (sometimes called a Restful API) is a kind of application programming interface (API or web API) that adheres to the REST architectural style and allows users to communicate with restful web services. Roy Fielding, a computer scientist, coined the term "REST" to describe representational state transfer [41].

2.14.1 A Difference Between Restful Web Service And an API

A RESTful API is a sort of application programming interface. REST services are APIs, but not all APIs are REST services.

An API is a general term. It refers to the way one piece of code interacts with another in general. In web development, the method through which we obtain data from an internet service is known as an API. The API documentation contains a list of URLs, query parameters, and additional instructions on how to use the API and what kind of answer you could receive for each query.

REST is a set of rules, standards, and laws for developing a web API. Because an API may be structured in a variety of ways, having an agreement approach saves time when making decisions when designing one and when learning how to use one. SOAP and GraphQL are two of the more popular API paradigms [42].

2.15 Types of Sculling

The two techniques to cope with system expansion are vertical and horizontal scaling [43].

- Vertical scaling refers to boosting the capacity of a single server by adding a more powerful CPU, more RAM, or more storage. Due to technological constraints, a single system may not be powerful enough to handle a certain demand. Furthermore, Cloud-based services have set limits depending on the hardware configurations available. As a consequence, vertical scaling has reached its practical limit.
- Horizontal scaling spreads the system's data and load over several servers, with more servers added as required to increase capacity. While a single machine's total speed or capacity may be limited, each machine only handles a piece of the total workload, possibly resulting in more efficiency than a single high-speed, high-capacity server. Adding several servers as required to increase deployment capacity will be less costly than purchasing high-end hardware for a single computer. However, the execution would be more difficult in terms of infrastructure and

maintenance. "MongoDB sharding enables horizontal scalability [44]" [45].

2.16 Postman

Postman is an API development tool that lets you design and consume APIs. Through Postman, cooperation is facilitated and streamlined at each stage of the API lifecycle, making it simpler to create better APIs. There's also an app for evaluating APIs. It's an HTTP client with an interface for testing HTTP requests, enabling us to acquire a variety of replies to verify [46]. This is a section describing the methods, response codes, environments, and key features in Postman:

A. Methods

Postman has a lot of endpoint interaction methods. Here are some of the most commonly used, as well as their functions:

1. GET: Obtain information.
2. POST: Fill up the blanks with relevant information.
3. PUT: Replace data.
4. PATCH: Ensure that some of the information is up to date.
5. DELETE: Delete the data.

B. Response Codes

When we use Postman to test APIs, we generally get a variety of response codes. The following are a few of the most popular, as shown in table 2.1.

Table 2.1 Table of Popular Response Codes

Series	Descriptions	Responses
100	Temporal responses	102 Processing
200	Responses where the client accepts the request and the server processes it successfully	200 Ok
300	Responses related to URL redirection	301 Moved Permanently
400	Client error responses, for instance	400 Bad Request
500	Server error responses	500 Internal Server Error

C. Environments

Postman also allows us to build various environments by generating/using variables, such as a URL variable for different test environments (Dev-QA), which allows us to run tests in different contexts using existing requests.

D. Key Features

Postman is a graphical user interface program that allows us to test APIs. Postman is an easy-to-use application that helps us save time when running tests.

2.17 Autocannon Benchmark

Production Environment: In the environment of production, there are tools based on this technology that can assist us in the production environment, allowing us to avoid doing the hard work [47].

Autocannon is a node-based HTTP/1.1 benchmarking tool that supports HTTP pipelining and HTTPS. As a result, autocannon can increase the server workload for each open connection. It is CPU-bound and written in JavaScript

for the Node.js runtime. The autocannon is a versatile weapon. You may use it as a command-line tool or directly to perform some complex testing scenarios. You may also customize it with a variety of different features and settings to meet your individual needs.

- Latency: A statistical object that contains response latency information.
- Throughput: A statistical object that contains information about the throughput of response data per second.
- Duration: The length of time it took to complete the exam in seconds.
- Connections: The number of connections that have been made (value of connections).
- Pipelining: The number of requests sent through a pipeline per connection (value of pipelining).
- URL: The URL that was chosen as the target.

Requests, latency, and throughput are objects with the following shape:

- Min: This statistic's lowest value.
- Max: This statistic's maximum value.
- Average: The mean (average) value.
- The standard deviation is referred to as Stddev.

One table is for request latency, while the other is for request volume.

As shown in figure 2.7, the request timings at the 2.5 percent percentile are the fast outliers at 50%, the median at 97.5%, the slow outliers, and at 99%, the extremely slowest outliers are included in the latency table. Lower means quicker in this case.

The request volume table shows how many requests were submitted and how many bytes were downloaded. Once per second, these values are sampled. More requests were handled if the value was higher. In the worst-case scenario,

2.29 MB was downloaded in 10 seconds (slowest 1 percent). The Min value, as well as the 1 percent and 2.5 percent percentiles, are all from the same sample because we only ran for 5 seconds. These values will differ much more over extended periods of time [47].

```
Running 10s test @ http://localhost:3000
10 connections

┌─────────┬──────┬──────┬───────┬──────┬─────────┬─────────┬──────────┐
│ Stat    │ 2.5% │ 50%  │ 97.5% │ 99%  │ Avg     │ Stdev   │ Max      │
├─────────┼──────┼──────┼───────┼──────┼─────────┼─────────┼──────────┤
│ Latency │ 0 ms │ 0 ms │ 0 ms  │ 1 ms │ 0.02 ms │ 0.16 ms │ 16.45 ms │
└─────────┴──────┴──────┴───────┴──────┴─────────┴─────────┴──────────┘

┌───────────┬─────────┬─────────┬─────────┬─────────┬──────────┬──────────┬─────────┐
│ Stat      │ 1%      │ 2.5%    │ 50%     │ 97.5%   │ Avg      │ Stdev    │ Min     │
├───────────┼─────────┼─────────┼─────────┼─────────┼──────────┼──────────┼─────────┤
│ Req/Sec   │ 20623   │ 20623   │ 25583   │ 26271   │ 25131.2  │ 1540.94  │ 20615   │
├───────────┼─────────┼─────────┼─────────┼─────────┼──────────┼──────────┼─────────┤
│ Bytes/Sec │ 2.29 MB │ 2.29 MB │ 2.84 MB │ 2.92 MB │ 2.79 MB  │ 171 kB   │ 2.29 MB │
└───────────┴─────────┴─────────┴─────────┴─────────┴──────────┴──────────┴─────────┘

Req/Bytes counts sampled once per second.

251k requests in 10.05s, 27.9 MB read
```

Figure 2.7 Result of Autocannon Benchmark Tool

When the -l option is used, a third table appears that displays all of the autocannon's latency percentiles, as shown in figure 2.8. If a large number of queries were issued, this could provide further information.

Percentile	Latency (ms)
0.001	0
0.01	0
0.1	0
1	0
2.5	0
10	0
25	0
50	0
75	0
90	0
97.5	0
99	1
99.9	1
99.99	3
99.999	15

Figure 2.8 All of the Autocannon's Latency Percentiles

2.18 Load Balancing Metrics

Various metrics considered in existing load balancing techniques are discussed below:

1. Throughput: this metric is used to determine how many requests have been completed. Megabytes per second (MB/s) is the most common unit of measurement for throughput. To enhance the system's performance, it should be set to a high value.
2. Performance: is used to evaluate the system's efficiency. It must be enhanced at a cost that is fair. Response times, for example, must be

shortened while acceptable delays are maintained. (It is represented by an average, percentage, maximum, and minimum number of requests).

3. Resource Utilization: This is a technique for monitoring how well resources are used. Its load balancing efficiency should be enhanced. (It is represented by single-threaded and multi-threaded).
4. Scalability: the ability of an algorithm to balance load across any number of nodes in a system. To deal well with a growing number of requests or to adapt to them. For example, as more resources are added, this term might refer to a system's capacity to increase its total output when it is overloaded. This measure needs improvement. (It is represented by throughput)
5. Fault tolerance: an algorithm's capacity to keep load balancing smooth in the event of a node or connection failure. Load balancing should be a trustworthy and fault-tolerant solution.
6. Latency: is the time it takes for data to travel across a network. It's commonly expressed as a round-trip delay, or the time it takes for data to travel from source to destination and back. As well as the time it takes for the request to be served. It's utilized for things like latency spikes and performance.
7. Response time: is defined as the time elapsed between a client initiating a request and receiving a response. In a distributed system, it is the time it takes for a particular load balancing method to respond. This parameter should be kept to a minimum.

2.19 Description of Autocannon Benchmark Metrics

In this section, explain the metrics that are utilized in the figures in Autocannon Benchmark.

A standard deviation is knowing the center of a data collection doesn't tell us much. We'd want to learn more about how our data sets are organized in general. The standard deviation is a measurement of how evenly a set of data is

spread. In business, standard deviations of price data are often used to assess vulnerability. The standard deviation is used to calculate the margin of error in opinion polls.

An average is a single number that represents a collection of numbers, often the sum of the numbers divided by the number of numbers in the collection. It represents the average number of requests per second.

A percentage is a number or percentage that is expressed as a fraction of 100. It is often denoted using the percent sign, "%". It represents the percentage of the number of requests during the specified time in seconds.

A maximum is the representation of the time spent waiting for a web request to be processed in latency.

A minimum is a representation of the number of web requests processed per second.

Single-threaded processes execute instructions in a single order. To look at it another way, only one command is handled at a time.

Multi-threaded processes are the reverse of single-threaded processes. Multiple components of a program can be executed concurrently using these methods.

Chapter Three

The Proposed System Architecture

Chapter Three
The Proposed System Architecture

3.1 Introduction

This thesis proposes a system architecture based on Docker, Nginx, Node JS, Redis, and MongoDB, as shown in figure (3.1), to provide high-performance for each request or response through an API, as well as the usage of caching and database for best high-performance.

The proposed system in this thesis is to build a pool of services using Docker swarm containers, then build a load balancer (using Nginx) and enhance an algorithm to distribute the load of network requests to a number of web services (using Node JS) after improving the processing in CPUs that use all of their resources to process the requests, and then find solutions to the problems that arise after using this system (by using Redis and MongoDB).

Figure (3.1) shows the description of the proposed system at zero step. The client sends some requests through the internet to the load balancer, which serves these requests. At step one, the load balancer acts as a gateway to receive and respond to requests from the client, enabling this request to be routed to the server selected depending on the enhanced algorithm. At step two, the request process is back-end served by Node JS. In step three, we improved processing by running Node JS in cluster mode by running one core as a master and the other core as a worker on the CPU of the server. In the fourth step, Redis services run as a cache to store any most-used value and solve the problem of separate memory storage. The final step is the fifth, the MongoDB service, which stores the data in the database as a document and streams video.

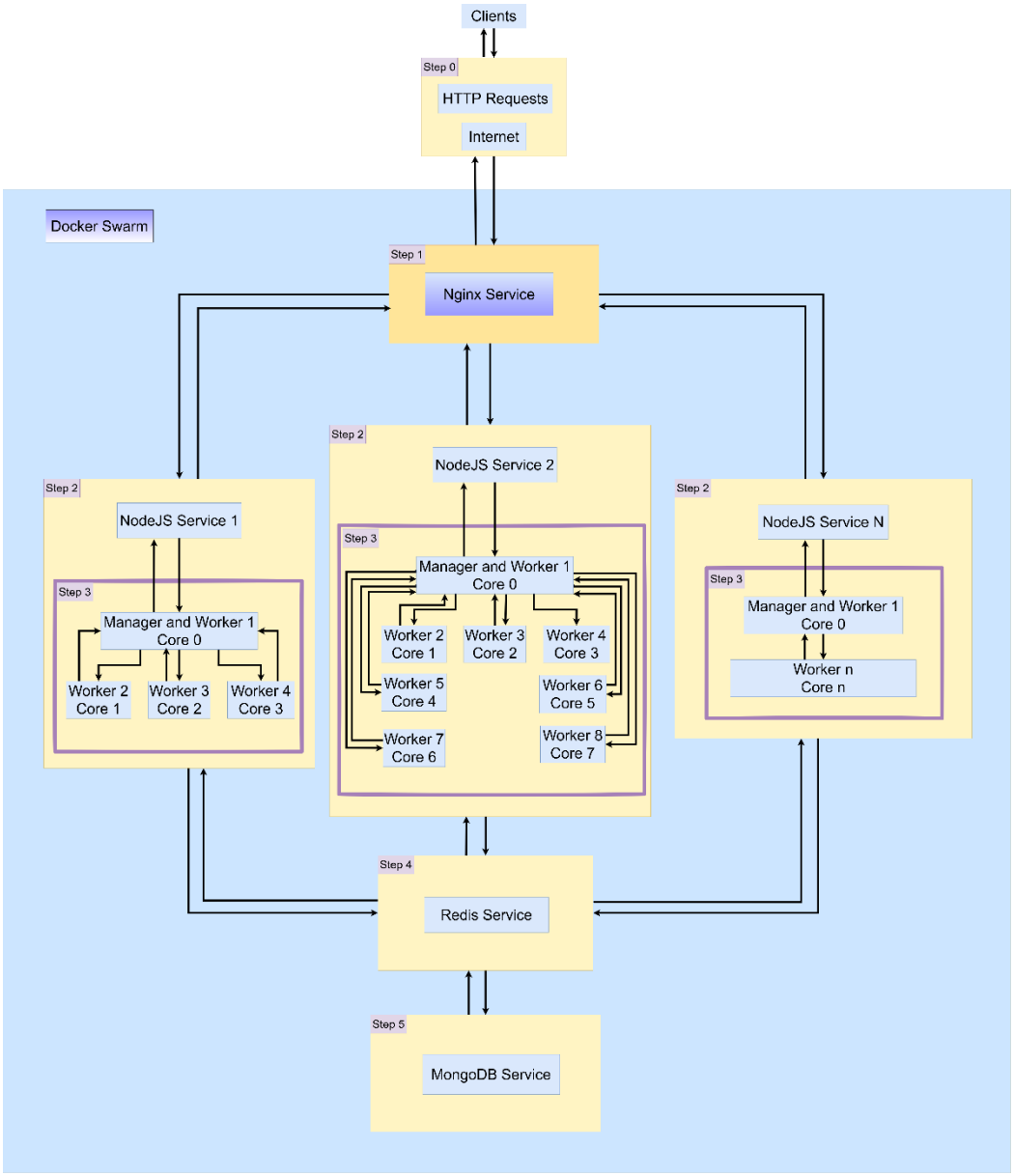


Figure 3.1 Steps Deploy Services of the Proposed System in Docker Swarm

3.2 The Proposed System

The proposed system consists of two stages, or parts. The first part enhances distributed load balancing algorithms. In the second part improve processing of the outcomes of distributed load balancing.

3.2.1 The First Part of the Proposed System

The first part of the proposed system implemented enhanced distributed load balancing with the default algorithms (Round Robin and Least Connection)

by selecting these algorithms and adding weight depending on the number of cores inside the processer to make the enhanced algorithms (Enhanced Weight Round Robin and Enhanced Weight Least Connection), which were implemented by using Nginx software as a service inside a Docker swarm. The main idea of this proposal is to enhance distributed or routed requests.

3.2.2 The Second Part of the Proposed System

In the second part of the proposed system, after defining and enhancing the algorithms. Now improving the results of distributed load balancing processing are fully optimized by using the Node JS cluster (Multi-Threading) to handle requests by using the full complete CPU (use all cores in the CPU) of the devices.

The cluster module in the proposed system allows you to create child processes that operate in parallel and share the same server port. To take advantage of the computer's multi-core systems, the Cluster module allows you to quickly establish child processes that each operate on their own single thread to manage the load.

For the cache memory, the load distributed on the database was implemented by Redis, which is used to shard data with one or a cluster of servers.

For the database, the load distribution on the database was implemented by MongoDB, which is used to shard data across many servers and increases the number of replication in a cluster.

3.3 The First Part Enhance Algorithms

Presentation of the enhance algorithm proposal and configuration of the Nginx service. Load balancing HTTP traffic among web or application server groups is executed using a number of methods. Load balancing among many application instances is a common approach for increasing throughput, decreasing latency, and ensuring fault tolerance. Load balancing and scaling

techniques provided by NGINX are used by users to create large-scale, highly available web services.

3.3.1 Choosing The Weighted

As you can see in chapter two, the Round Robin approach is very simple. It will fail in some situations.

What if Server 1 has a faster processor, more RAM, and other features than Server 2? Isn't it true that Server 1 should have a larger traffic capacity than Server 2? Unfortunately, a Round Robin load balancer will not be able to tell the difference between the two servers. Ignoring the fact that it doesn't matter that the two servers have different abilities, the load balancer will redistribute requests equally. Server 2 may get overwhelmed more quickly as a consequence, leading to its crash.

The Round Robin algorithm works well (but not effectively) on clusters of servers with similar characteristics.

- **Drawback Of Round Robin Algorithm (RR)**
  1. The overload of context switching is a term that refers to not taking past loads into consideration.
  2. The failure to consider the various server characteristics.
- **Drawback Of Least Connections Algorithm (LC)**
  1. The failure to consider the various server characteristics.

Finally, solutions the purpose is selected weighted depending on the number of cores in the CPU of each server.

3.3.2 Enhanced Weighted Round Robin (EWRR)

When Server 1 has better specifications than Server 2, you might want to use an algorithm that distributes more requests to the server that can handle more load. One such algorithm is the Enhanced Weighted Round Robin.

The Enhanced Weighted Round Robin is comparable to the Round Robin in that the requests are dispersed cyclically among the services, but there is one distinction. A greater number of requests will be sent to the server that has the highest characteristics.

So how would the load balancer determine which node is more powerful? Simple put, you tell it ahead of time. Basically, you assign "weights" to each node when you set up the load balancer. Of course the node with the stronger requirements, should be given more weight. Weights are frequently indicated as a number of actual capacity. If Server 1 has 5x the capacity of Server 2, for example, give it a weight of 5 and Server 2 a weight of 1.

An example of a scenario in which two nodes receive the twelfth number of requests. The first is weighed to equal 5, and the second is weighed to equal 1 as show in table 3.1. As a result, show in table 3.2, node 1 will receive the first five requests while node 2 will receive the sixth. The same method will be followed if more requests arrive. The seventh, eighth, ninth, tenth, and eleventh will all be assigned to node 1, while the twelfth will be sent to node 2.

Table 3.1 Node with deferent number of weighted

Host Name	Weighted
Node 1	5
Node 2	1

Table 3.2 Distributed requests coming by the EWRR algorithm

Server	Weight	EWRR Algorithm
Node 1	5	1, 2, 3, 4, 5, 7, 8, 9, 10, 11
Node 2	1	6, 12

The algorithm Enhanced Weighted Round Robin (EWRR) was chosen for more than its capacity. For example, you could want to utilize it if you want one server to get fewer connections than another equally competent server, since the first is that you're operating mission-critical services and don't want it to get overloaded extremely early. When servers have varying degrees of computing capability, this is a useful model to utilize.

Algorithm 3.1 below represents the Enhanced Weighted Round Robin (EWRR) algorithm in a program with code in C++. To represent how a request is routed to the next server depending on the weight of the server.

Algorithm 3.1 : Enhance Weight Round Robin (EWRR)

```
1  #include <stdio.h> // Library
2  #include<conio.h> // Library
3  #include <iostream> // Library
4  #include <string> // Library
5  using namespace std; // Library
6
7  int requests = 12; // Number Of Requests Come From Clients
8  const int node = 2; // Number Of Nodes
9  #define MAX 50  // Maximum Size Of Queue
10 int intArray[MAX]; // Maximum Size Of Queue
11 int front = 0; // Index For First Queue
12 int rear = -1; // Index For Last Queue
13 int itemCount = 0; // Length Of Elements In Queue
14 int a = 0; // Begin At The Top Of The List With The First Node
15
16 // Check If Queue Full
17        bool isFull() {
18        return itemCount == MAX;
19 }
20
21 // Check If Queue Empty Or Not Return (1 = True) (0 = False )
22 bool isEmpty() {
23        return itemCount == 0;
24 }
```

```
25
26  // Remove One Item From Queue
27  int pop() {
28         int data = intArray[front++];
29         if(front == MAX){
30                front = 0;
31         }
32         itemCount--;
33         return data;
34  }
35
36  // Insert New Element In Queue
37  void push(int data) {
38         if(!isFull()) {
39                if(rear == MAX-1) {
40                       rear = -1;
41                }
42                intArray[++rear] = data;
43                cout<<data<< " Pushed into Queue\n";
44                itemCount++;
45         }else{
46                cout<<"Queue Overflow\n";
47         }
48  }
49
50  int main() {
51
52         // Specify Size Of The Array Nodes
53         std::string matrix [node][2];
54
55         // Get Size Of Node Array
56         #define my_sizeof(type) ((char *)(&type+1)-(char*)(&type))
57         int size = my_sizeof(matrix)/my_sizeof(matrix[0]);
58
59         // Input Nodes In List Of Array
60         for(int i=0;i<node;i++){
61                string text_string = to_string(i+1);
62                // [Server Node]     [Weight]
63                matrix [i] [0] = "Node" + text_string;
64                // Enter Number Value Weight Of Node
65                cout<<"Enter Weight Of "<<matrix [i][0]<<" :";
66                string x;
67                cin>>x;
68                matrix [i] [1] = x;
69         }
70
71         // Print The Node And Weight
72         for(int i=0;i<node;i++){
73                cout<<matrix[i][0]<<" "<<matrix[i][1]<<"\n";
74         }
75
76         // Request Pushed Into Queue
77         for(int k=1;k<=requests;k++){
78                push(k);
79         }
80
81         // Node Defien Variable As Pointer And Weight
82         int list_node = 0;
83         int w =atoi(matrix[list_node][1].c_str());
84
85         // Routed Request N To Node N
86         for (int i=1;i<=requests;i++){
```

```
87                  // If The Weight Of Node Is Greater Than Zero, Then Continue
88                  if(w>0){
90                          /* If There Are Any Requests In The Queue,
91                           They Will Be Routed */
92                          if(!isEmpty()){
93                                  cout<<"Request "<<pop()<<"    "
94                                          <<matrix[list_node][0]<<"\n";
95                          }
96                          /* Then, After The Request Is Routed,
97                           Make The Weight Of This Node Lower By One */
98                          w = w -1;
99                  }else{
100                         /* If You Reach The End Of The List Of Nodes,
101                          Return To The Beginning And Continue
102                          Routing Requests */
103                         if(list_node==size-1){
104                                 list_node = 0;
105                                 w =atoi(matrix[list_node][1].c_str());
106                                 if(!isEmpty()){
107                                         cout<<"Request "<<pop()<<"    "
108                                                 <<matrix[list_node][0]<<"\n";
109                                 }
110                                 w = w -1;
111
112                         }else{
113                                 /* If You Reach This Point Of Code, It Means
114                                  The Weight Of This Node Is Equal To Zero.
115                                  Go To The Next Node */
116                                 list_node = list_node + 1;
117                                 w =atoi(matrix[list_node][1].c_str());
118                                 if(!isEmpty()){
119                                         cout<<"Request "<<pop()<<"    "
120                                                 <<matrix[list_node][0]<<"\n";
121                                 }
122                                 w = w -1;
123                         }
124                 }
125         }
126 }
```

To show this algorithm's simple design, you can go to figure 3.2 to see the flowchart of the Enhance Weight Round Robin Algorithm (EWRR).

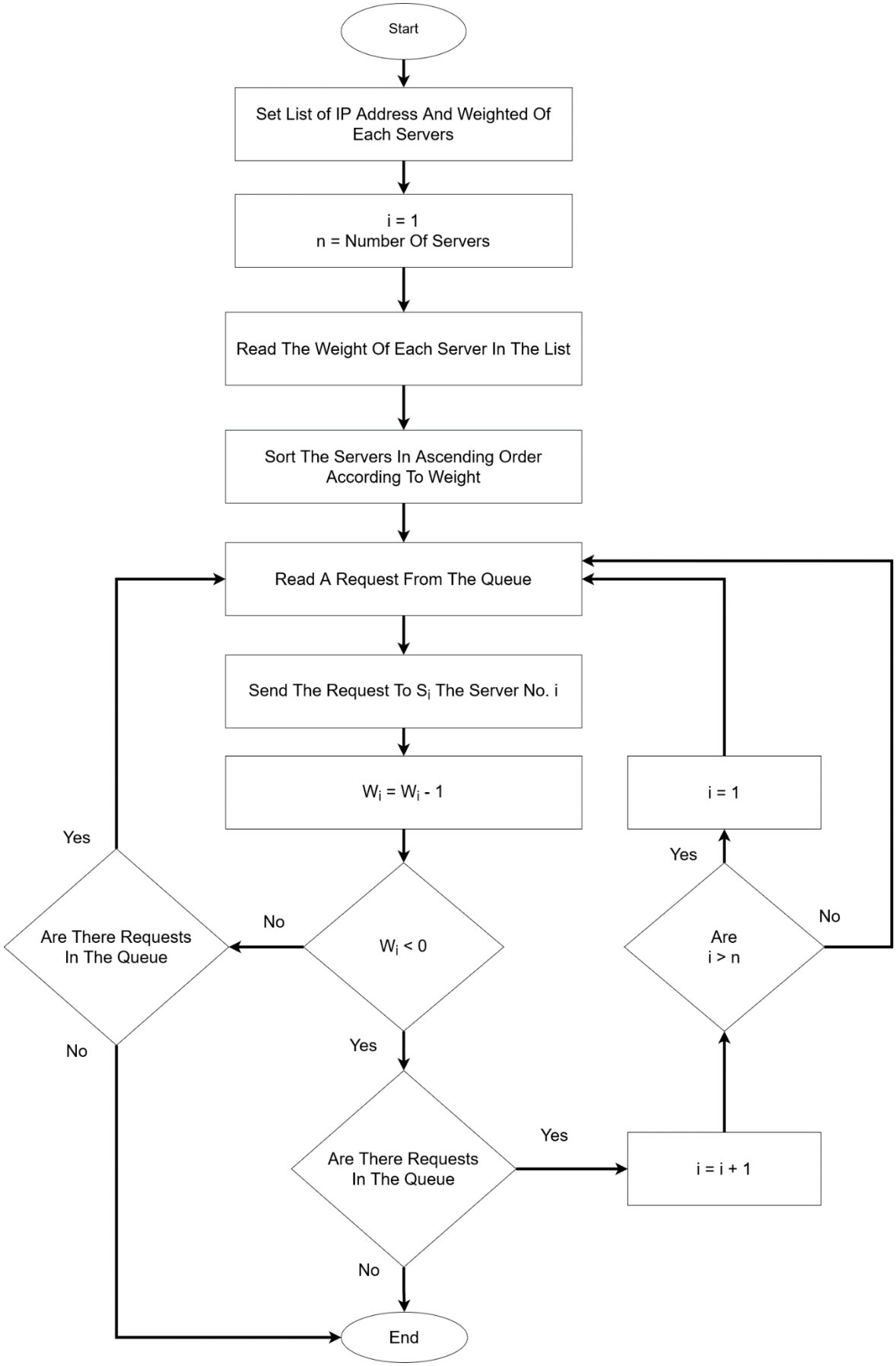


Figure 3.2 Flowchart of Enhance Weight Round Robin Algorithm (EWRR)

3.3.3 Enhanced Weighted Least Connections (EWLC)

When it comes to least connections, the Enhanced Weighted Least Connections algorithm performs what the Enhanced Weighted Round Robin algorithm does for Round Robin. As a consequence, it includes a "weight" component dependent on the number of servers each can support. You'll have to choose each server's "weight" before the game begins, just as in the Enhanced Weighted Round Robin.

A load balancer that employs the Enhanced Weighted Least Connections approach now takes into consideration the weights (capabilities of each server) , as well as the actual number of clients connecting to each server.

The Enhanced Weighted Least Connections algorithm keeps track of the number of active connections on each application server. Based on the following combination, the service transfers a new connection to a server:

- It has a high number of core CPU weights.
- The number of active connections that it has.

This algorithm takes longer to compute than the algorithm with the least connection. On the other hand the extra computation, results in the traffic being distributed more effectively to the server that is best suited to handle the request.

Enhanced Weighted Least Connections: Servers are weighted according to their processing capacity or on the number of cores if they have the same ability.

The number of active connections on each server, as well as the relative capacity of the servers, are used to spread the load.

This algorithm is for application servers only, not authentication or authorization servers, and it necessitates the use of the Application Optimization feature. We use Redis servers to manage this.

When servers have variable degrees of processing capability, this is a suitable method to utilize. This algorithm distributes traffic in concurrency because this is the solution to the problem of processing huge traffic at the same time.

Algorithm 3.2 below represents the Enhanced Weighted Least Connections (EWLC) algorithm in a program with code in C++. To represent how a request is routed to the next server depending on the overhead of the server.

Algorithm 3.2 : Enhance Weight Least Connection (EWLC)

```
1  #include <stdio.h> // Library
2  #include<conio.h> // Library
3  #include <iostream> // Library
4  #include <string> // Library
5  using namespace std; // Library
6
7  int requests = 20; // Number Of Requests Come From Clients
8  const int node = 4; // Number Of Nodes
9  #define MAX 50  // Maximum Size Of Queue
10 int intArray[MAX]; // Maximum Size Of Queue
11 int front = 0; // Index For First Queue
12 int rear = -1; // Index For Last Queue
13 int itemCount = 0; // Length Of Elements In Queue
14 int a = 0; // Begin At The Top Of The List With The First Node
15
16 // Check If Queue Full
17        bool isFull() {
18        return itemCount == MAX;
19 }
20
21 // Check If Queue Empty Or Not Return (1 = True) (0 = False )
22 bool isEmpty() {
23        return itemCount == 0;
24 }
25
26 // Remove One Item From Queue
27 int pop() {
28        int data = intArray[front++];
29        if(front == MAX){
30               front = 0;
31        }
32        itemCount--;
33        return data;
34 }
35
36 // Insert New Element In Queue
37 void push(int data) {
38        if(!isFull()) {
39               if(rear == MAX-1) {
40                      rear = -1;
41               }
42               intArray[++rear] = data;
43               cout<<data<< " Pushed into Queue\n";
44               itemCount++;
45        }else{
```

```
46                cout<<"Queue Overflow\n";
47        }
48  }
49
50  int main() {
51
52        // Specify Size Of The Array Nodes
53        std::string matrix [node][4];
54
55        // Get Size Of Node Array
56        #define my_sizeof(type) ((char *)(&type+1)-(char*)(&type))
57        int size = my_sizeof(matrix)/my_sizeof(matrix[0]);
58
59        // Input Nodes In List Of Array
60        for(int i=0;i<node;i++){
61              string text_string = to_string(i+1);
62              // [Server Node]     [Weight]      [Current Connection] [Overhead]
63              matrix [i] [0] = "Node" + text_string;
64              // Enter Number Value Weight Of Node
65              cout<<"Enter Weight Of "<<matrix [i][0]<<" :";
66              string x;
67              cin>>x;
68              matrix [i] [1] = x;
69              // Enter Number Value Current Connection Of Node
70              cout<<"Enter Current Connection Of "<<matrix [i][0]<<" :";
71              string y;
72              cin>>y;
73              matrix [i] [2] = y;
74              matrix [i] [3] = "0";
75        }
76
77        // Print The Node And Weight And Current Connection
78        for(int i=0;i<node;i++){
79              cout<<matrix[i][0]<<" "<<matrix[i][1]<<" "<<matrix[i][2]<<"\n";
80        }
81
82        // Request Pushed Into Queue
83        for(int k=1;k<=requests;k++){
84              push(k);
85        }
86
87        // Define The Basic Variables Of The Algorithm
88        double Current_Connections = 0;
90        double Weighted = 0;
91        double Overhead = 0;
92        double min = 0;
93        // Node Defien Variable As Pointer
94        int i_matrix_in_number=0;
95
96        // Routed Request N To Node N
97        for(int j=1;j<=requests;j++){
98
99              // This Is For Calculating The Overhead For Each Node In An Array
100             for(int d=0;d<size;d++){
101                   Current_Connections = atoi(matrix[d][2].c_str());
102                   Weighted = atoi(matrix[d][1].c_str());
103                   Overhead = Current_Connections / Weighted;
104                   std::string String_Overhead = std::to_string(Overhead);
105                   matrix[d][3]=String_Overhead;
106                   cout<<"Overhead:( "<<matrix[d][3]<<" ) \n";
107             }
108
```

```
109                 /* "min" Is Value That Storage The Lowest Number
110                  Of Overhead Of Nodes */
111                 Current_Connections = atoi(matrix[0][2].c_str());
112                 Weighted = atoi(matrix[0][1].c_str());
113                 Overhead = Current_Connections / Weighted;
114                 min = Overhead;
115
116                 /* All Nodes Must Be Tested In Order To Achieve A Low Minimum
117                  Current Connection And Storage Value */
118                 for(int i=size;i>=0;i--){
119                               Current_Connections = atoi(matrix[i][2].c_str());
120                               Weighted = atoi(matrix[i][1].c_str());
121                               Overhead = Current_Connections / Weighted;
122                               if(Overhead < min){
123                                     min = Overhead;
124                                     i_matrix_in_number=i;
125                               }else if(Overhead <= min){
126                                     i_matrix_in_number=i;
127                               }
128                 }
129
130                 /* The Request Is Routed To The Node With The
131                  Lowest Overhead And Highest Weight
132                  If The Two Nodes Are Equal */
133                 cout<<"Request: "<<pop()
134                       <<" Name_Host: "<<matrix[i_matrix_in_number][0]
135                       <<" Current_Connections: "<<matrix[i_matrix_in_number][2]
136                       <<"\n";
137
138
139                 /* Then, After The Node Has Selected And Routed The Request,
140                  Make The Current Connection Of This Node Increase By One */
141                 int s = atoi(matrix[i_matrix_in_number][2].c_str())+1;
142                 std::string ss = std::to_string(s);
143                 matrix[i_matrix_in_number][2]=ss;
144                 cout<<"The New Value Of Current Connections: "
145                       <<matrix[i_matrix_in_number][2]<<"\n";
146                 cout<<"--------------------\n";
147           }
148 }
```

To show this algorithm's simple design, can go to figure 3.3 to see the flowchart of the Enhance Weight Least Connection Algorithm (EWLC).

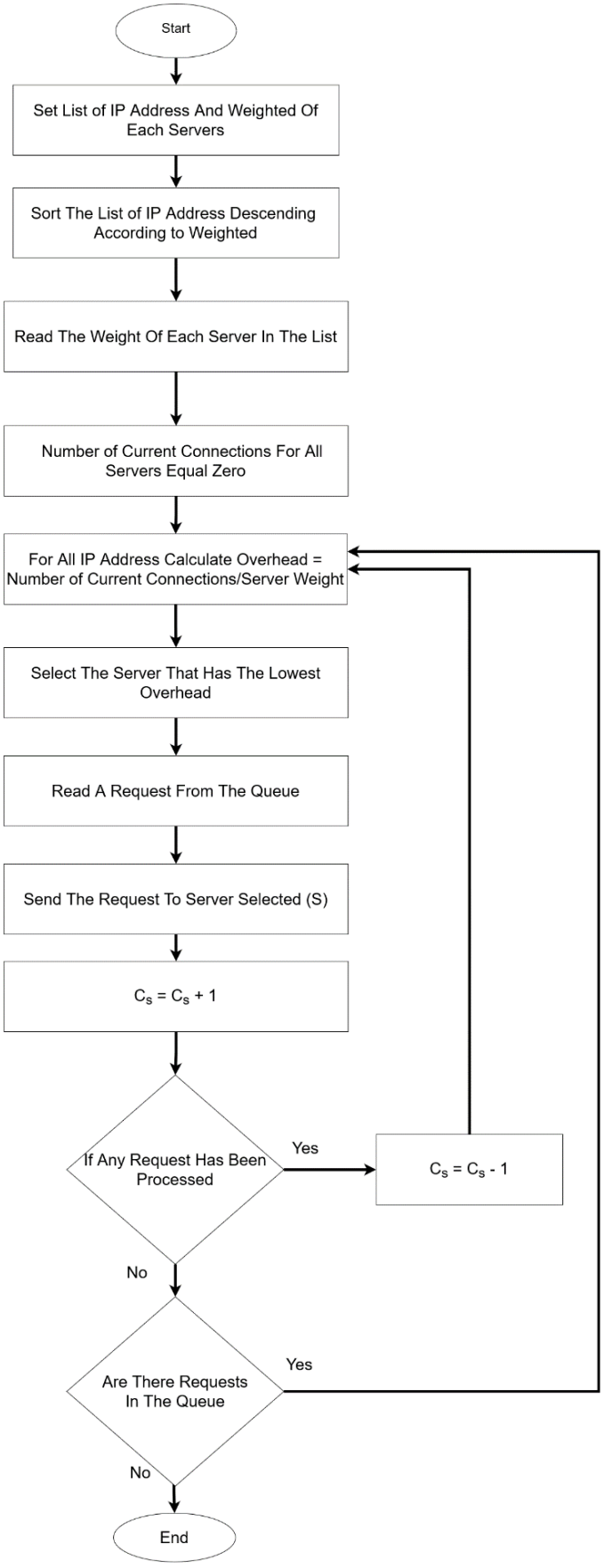


Figure 3.3 Flowchart of Enhance Weight Least Connection Algorithm (EWLC)

3.3.4 Backend Calculations Of The EWLC

A number between 0 and 100 can be assigned to each backend server to represent the volume of requests it will receive. Even if the backend server is regarded as healthy, requests will not be routed to it if its weight is 0.

Increase the number of Enhance Weighted Least Connections: If none of the servers has a weight of 0, the load balancer uses the formula to compute the load on each backend server:

Overhead = Number of current connections / Server weight.

Server weight = The number of cores in a central processing unit

Overhead is calculated as the number of current connections divided by the server's weight.

The number of existing central processing units in a single computer device (die or chip) is referred to as “cores”.

In each request distribution, the load balancer distributes requests to the backend server with the lowest overhead.

Summary: To distribute particular amounts of traffic to accessible destinations within a pool, use weighted load balancing. Traffic will be reworked to available destinations based on their relative weights.

Case study example: We have 4 Server with different numbers of cores in the CPU chip as shown in table 3.3.

Table 3.3 Node with deferent number of core

Host Name (Processor)	Total Cores
Node 1 (Core i7)	8
Node 2(Core i5)	6
Node 3 (Core i3)	4
Node 4 (Core i2)	2

With 20 requests coming in at the same time, the EWLC algorithm is working as shown in table 3.4.

Table 3.4 Distributed requests coming by the EWLC algorithm

Server	Weight	EWLC Algorithm
Node 1	8	1, 5, 7, 10, 11, 15, 17, 20
Node 2	6	2, 6, 9, 12, 16, 19
Node 3	4	3, 8, 13, 18
Node 4	2	4 , 14

In table 3.3, distributed requests are made by reading the weights of all nodes and current connections and then selecting the one with the highest weight and low current connection depending on the calculation of overhead.

An overload value is used in an algorithm to assist in selecting which server receives the next request. A lower-overload server receives more traffic

than a higher-overload server. The traffic share of each server generally depends on its weight and current connection.

This is an explanation of how requests are routed in table 3.4 of the example of using the EWLC algorithm to distribute 20 requests.

C = Number of current connections

W = Number of cores

O = Overhead = Number of current connections / Server weight

Table 3.5 This is an example of distributing 20 requests using the EWLC algorithm

Host	C	W	O	Request 1	Host	C	W	O	Request 11
Node 1	0	8	0.000		Node 1	4	8	0.500	
Node 2	0	6	0.000		Node 2	3	6	0.500	
Node 3	0	4	0.000		Node 3	2	4	0.500	
Node 4	0	2	0.000		Node 4	1	2	0.500	
Requests	19				Requests	9			
Host	C	W	O	Request 2	Host	C	W	O	Request 12
Node 1	1	8	0.125		Node 1	5	8	0.625	
Node 2	0	6	0.000		Node 2	3	6	0.500	
Node 3	0	4	0.000		Node 3	2	4	0.500	
Node 4	0	2	0.000		Node 4	1	2	0.500	
Requests	18				Requests	8			
Host	C	W	O	Request 3	Host	C	W	O	Request 13
Node 1	1	8	0.125		Node 1	5	8	0.625	
Node 2	1	6	0.167		Node 2	4	6	0.667	
Node 3	0	4	0.000		Node 3	2	4	0.500	
Node 4	0	2	0.000		Node 4	1	2	0.500	
Requests	17				Requests	7			
Host	C	W	O	Request 4	Host	C	W	O	Request 14
Node 1	1	8	0.125		Node 1	5	8	0.625	
Node 2	1	6	0.167		Node 2	4	6	0.667	
Node 3	1	4	0.250		Node 3	3	4	0.750	
Node 4	0	2	0.000		Node 4	1	2	0.500	
Requests	16				Requests	6			
Host	C	W	O	Request 5	Host	C	W	O	Request 15
Node 1	1	8	0.125		Node 1	5	8	0.625	
Node 2	1	6	0.167		Node 2	4	6	0.667	

Node 3	1	4	0.250		Node 3	3	4	0.750	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	15				Requests	5			
Host	C	W	O	Request 6	Host	C	W	O	Request 16
Node 1	2	8	0.250		Node 1	6	8	0.750	
Node 2	1	6	0.167		Node 2	4	6	0.667	
Node 3	1	4	0.250		Node 3	3	4	0.750	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	14				Requests	4			
Host	C	W	O	Request 7	Host	C	W	O	Request 17
Node 1	2	8	0.250		Node 1	6	8	0.750	
Node 2	2	6	0.333		Node 2	5	6	0.833	
Node 3	1	4	0.250		Node 3	3	4	0.750	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	13				Requests	3			
Host	C	W	O	Request 8	Host	C	W	O	Request 18
Node 1	3	8	0.375		Node 1	7	8	0.875	
Node 2	2	6	0.333		Node 2	5	6	0.833	
Node 3	1	4	0.250		Node 3	3	4	0.750	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	12				Requests	2			
Host	C	W	O	Request 9	Host	C	W	O	Request 19
Node 1	3	8	0.375		Node 1	7	8	0.875	
Node 2	2	6	0.333		Node 2	5	6	0.833	
Node 3	2	4	0.500		Node 3	4	4	1.000	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	11				Requests	1			
Host	C	W	O	Request 10	Host	C	W	O	Request 20
Node 1	3	8	0.375		Node 1	7	8	0.875	
Node 2	3	6	0.500		Node 2	6	6	1.000	
Node 3	2	4	0.500		Node 3	4	4	1.000	
Node 4	1	2	0.500		Node 4	2	2	1.000	
Requests	10				Requests	0			

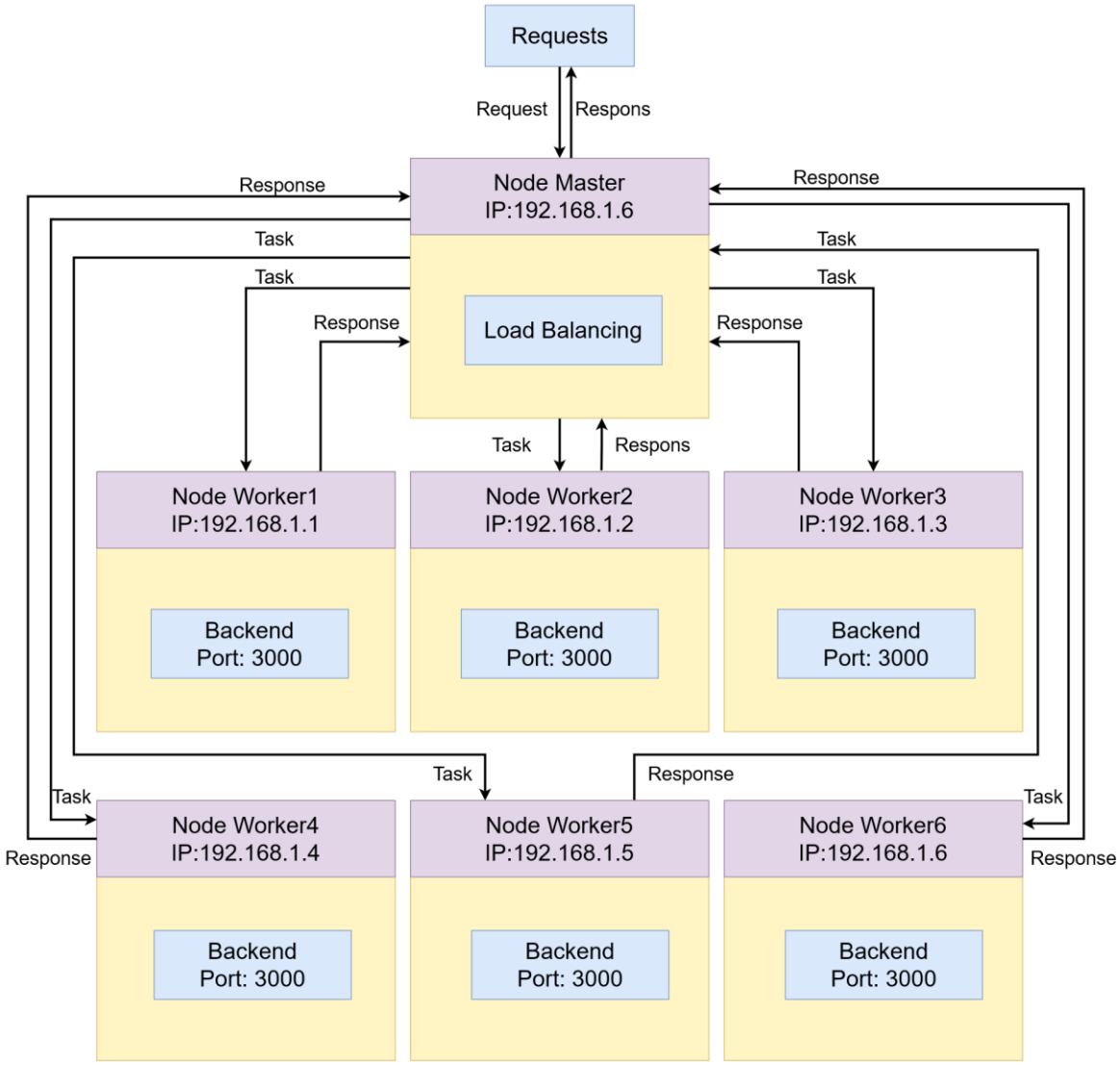


Figure 3.4 Design how web requests of clients are distributed in the proposed system

Figure 3.4 depicts how services are distributed in the proposed system, which includes seven services: one for load balancing and six for backend processing. Table 3.6 shows all the information about this service in the proposed system.

Table 3.6 Information about each service in the docker swarm of the proposed system

Host Name	Device	IP Address	Port	Subnet Mask	Gateway
Node Worker 1	Web Server (Backend)	192.168.1.1	3000	255.255.255.0	192.168.1.6:80

Node Worker 2	Web Server (Backend)	192.168.1.2	3000	255.255.255.0	192.168.1.6:80
Node Worker 3	Web Server (Backend)	192.168.1.3	3000	255.255.255.0	192.168.1.6:80
Node Worker 4	Web Server (Backend)	192.168.1.4	3000	255.255.255.0	192.168.1.6:80
Node Worker 5	Web Server (Backend)	192.168.1.5	3000	255.255.255.0	192.168.1.6:80
Node Worker 6	Web Server (Backend)	192.168.1.6	3000	255.255.255.0	192.168.1.6:80
Node Master	Load Balancer (Router)	192.168.1.6	80	255.255.255.0	---

3.4 The Second Part Improve Processing

Using a Node JS application as a cluster (multi-thread) to run on multiple cores of a CPU. Even when a single instance of a Node JS application runs on only one thread, multi-core CPUs aren't fully used. There will be occasions when you want to run a Node JS cluster of processes on a desktop workstation or a production server to make use of each CPU core. Cluster is a Node JS plugin that lets us execute a Node JS application across all of the CPU cores on a computer.

The cluster module allows Node JS to take advantage of a CPU's full capacity. By allowing us to set up child processes quickly. It also works hard in the background to communicate between the master and worker processes. As a result of all of this, the Node JS application's throughput will improve with each additional instance produced.

I. Work as an example

For example, if your server has 8 cores, to take advantage of all the core boxes, you have two options:

1. Node JS could start-up child processes or send messages to other worker processes for large-scale computation activities like image encoding. In this system, one thread will manage the flow of events, while N processes do heavy computing work and use the remaining 7 CPUs.
2. One can operate many Node JS servers on one computer, one for each core, to scale throughput on a web service. Requests for traffic should be shared between them. This gives great CPU ability to bind and allows for practically linear scaling of throughput as the number of cores increases.

II. Scaling a web service's throughput

Since Node JS version 0.6.0, the cluster module has been provided, making it simple to set up numerous node workers that may listen on a single port. It performs well and can easily scale its throughput on a multi-core machine.

One option is to run numerous instances of Node JS on the server and then place a load balancer in front of them like Nginx.

Single ports have some advantages (less coupling between processes, more complex load-balancing decisions), but they are more difficult to set up, and a built-in cluster module is a reduced option that most users prefer.

Because JavaScript is a single-threaded language, it only has a single call stack and memory heap. Node JS is a server-side programming language that works in the same way as JavaScript. It just uses one CPU core, regardless of how many CPU cores your system or cloud virtual machine has. The single-threaded nature of JavaScript is beneficial in a browser-based system, but it falls short in a backend system, that requires all of the machine's capabilities to avoid wasting resources.

However, Node JS is written in C, which comes with all of the necessary tools to create a multi-threaded application. Scalability with Node JS isn't a reconsideration. The Cluster module, which may use all of a machine's CPU

cores, is incorporated into the runtime. We can use the built-in Cluster module to scale a Node JS application over all of a machine's CPU cores. This ensures the system's high availability. Containers and horizontal scaling solutions should be considered for greater production-grade scalability.

III. Switch to Cluster-Mode

A cluster of node processes may be launched using the cluster module, which was introduced around version 0.6.0. The master process can fork and start additional child processes, all of which operate concurrently (parallel).

IV. A high level of availability

In the case of a crash or when we push new code, we must restart a single instance of the Node JS server. The solution to this problem is to run the app in many processes. We can just fork a new process if one crashes.

V. Alerts

The cluster module is a great method to grow your Node JS application, but there are a few things to keep in mind.

- No in-memory caching: Because numerous processes are executing in parallel, we simply will not be able to cache and retrieve some things in memory. Because there isn't any shared memory across processes. Therefore, I think, it is helpful because specialized caching methods trump in-memory caching. Now we would have to employ a caching system like Redis, which we would have to be able to access from any process.
- No Stateful Communication: Stateful network calls will no longer operate because communication will not always be with the same worker. As a result, sessions would be useless. A stateless token-based authentication solution such as JWT is preferable (JSON Web Token). This, I believe, is also advantageous, as it effectively makes our server stateless.

3.4.1 Deploy Node JS Services as a Cluster

In figure 3.5, a single thread executes a single Node JS instance. The user may create a cluster of Node JS processes to handle the request to take advantage of multi-core CPUs. You may quickly create child processes that share the same server ports by using the cluster module.

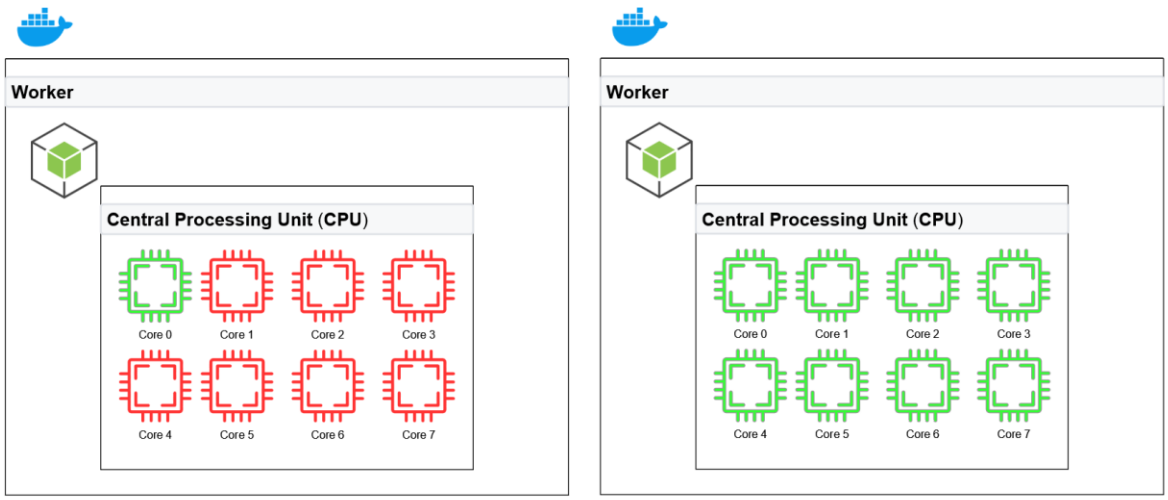


Figure 3.5 Node JS Run with CPU in (Default Single Core in First Figure and Multiple Threads Core in Second Figure)

The “process.fork()” child approach is used to establish worker processes that may communicate with the parent through IPC (instructions per cycle/clock). The number of things a CPU can perform in a single cycle is measured as IPC. Also, send and receive server handles. The cluster module can be used to disperse incoming connections.

This method (round-robin) is used on all systems. The central process listens to a single port, takes connections, and distributes them to the workers in a sequential order, as shown in the figure 3.6 below.

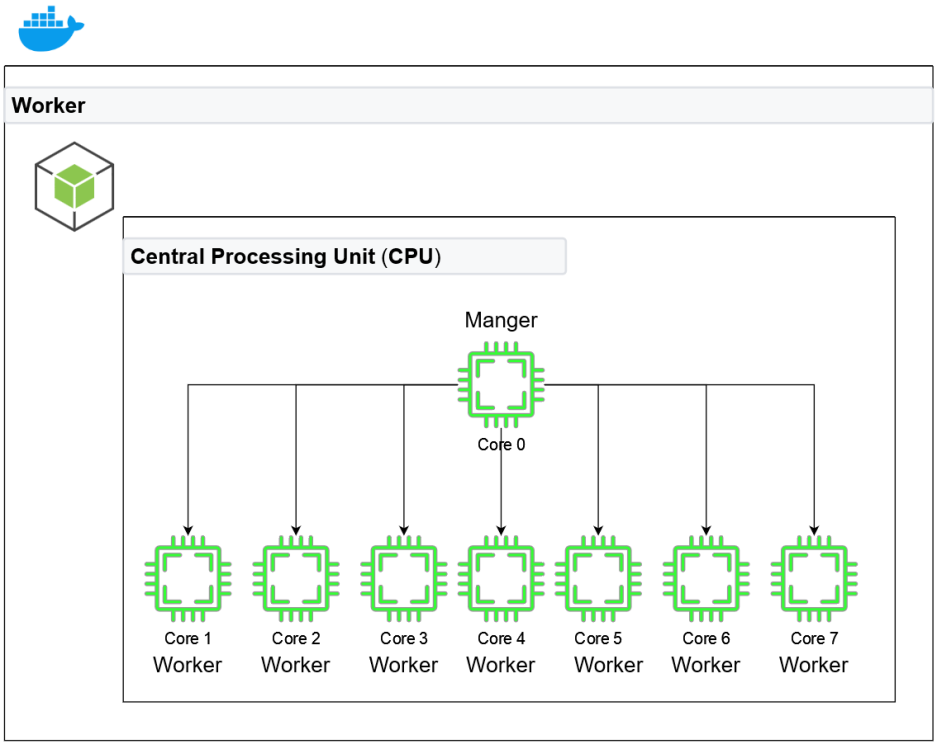


Figure 3.6 Node JS Design Work as Cluster (Multiple Threads Core)

Routing logic is not available in Node JS. When it comes to topics such as sessions and authentication, it's important to create an application that isn't dependent on in-memory data objects.

Because they are all independent and separate processes, a worker may be terminated or reproduce depending on program requirements without affecting other workers as show in figure 3.7. The server will accept connections as long as some workers are still active. If no workers are available, the current connection will be terminated, and new connections are blocked. Although networking is one of the cluster module's most frequent applications, it may also be used for other tasks that need worker processes.

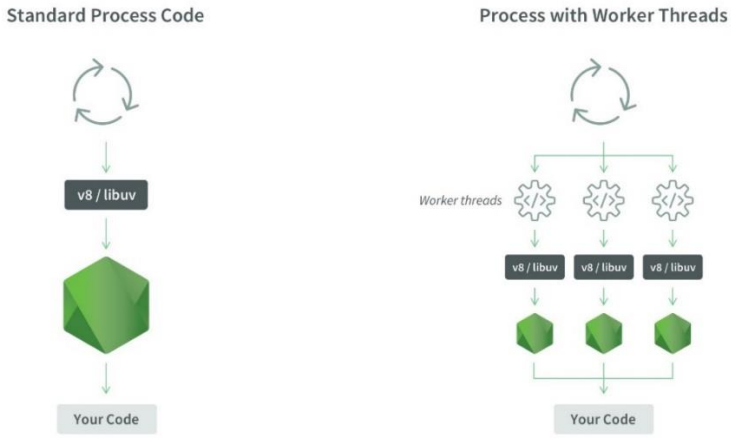


Figure 3.7 Differences in Node JS Design Work (Single Core vs Multiple Threads Core)

3.5 Session State And Redis

To solve the problem of no buffered data, the proposed system uses a service to cache any data, so for that purpose, the proposed system uses Redis for caching.

The Redis session cache is most typically used in scenarios where client requests are sent to two or more replicated web servers via a load balancing method. Session state is information about a user's current activity with an application, such as a website or a game. For as long as the user is signed in, a typical web application keeps a session for each connected user. Apps employ session states to remember user identification, login credentials, customization information, recent activities, shopping cart, and other information.

Reading and writing session data must be done without affecting the user experience at each user action. The session state is cached data for a given user or application that enables quick responses to user activities behind the application. As a result, no round-trip to the central database should be required while the user session is active.

When the user is disconnected, the session state life cycle comes to an end. Some information will be saved in the database for future use, but temporary data will be removed after the session is over.

3.6 Load Balancing With MongoDB

The proposed system, suggests enhancing algorithms, improving processing, and solving problems through caches at the end of the system's growth. The problem is that the database is an important part of this research. It was suggested to use MongoDB to make shards of data and many replications of them to make the system more available if it grows. The one of most powerful functions in MongoDB is GridFS, which divides files into chunks. This is important for streaming video.

The proposed streaming video with MongoDB uses NodeJS and GridFS library to store file videos in MongoDB as chunks and if any client requests this file video, it is returned as the part is needed depending on which click on the timeline of the video and depending on the header range of video to get this part of the chunk from the database (MongoDB) to the client and play the video like a movie.

MongoDB, allows data-driven developers to build rapidly, change quickly, and grow reliably. Let's have a look at what it means:

- Flexible Document Schemas: MongoDB's flexibility is critical for dealing with real-world data and adapting to changing requirements or environments.
- Powerful querying and analytics: MongoDB Query Language (MQL) JavaScript is used to create MongoDB queries. The language is quite simple to learn, and there are several tools for querying MongoDB data using SQL syntax. When querying data, you have a huge number of

choices, operators, expressions, and filters to choose from. You can do complicated analytics pipelines and query deep into documents.

- Scaling out horizontally is simple: MongoDB was designed to be a distributed database, making horizontal scaling simple. To maintain performance and expand horizontally, create clusters with real-time replication and shard large-to-high-throughput collections over many clusters.
- GridFS is a MongoDB standard for storing and retrieving large files, including photos, audio, and video. It saves files in a file system-like format, but the data is kept in MongoDB collections. GridFS has the capability of storing files larger than the 16MB document limit. GridFS divides a file into chunks and stores each chunk of data in its own document (up to 255 megabyte in size). It's simpler to access select areas of a file since files are separated into smaller chunks, avoiding the memory-intensive operation of loading the full file.

Chapter Four

Results And Discussion

Chapter Four

Results And Discussion

4.1 Introduction

The outcomes of the proposed system presented in chapter three are presented in this chapter, along with a discussion of the results.

4.2 Implementation

This section contains the explanation of the implementation of the proposed system, and after that, an implementation of the Docker swarm and the services it deploys in nodes.

4.2.1 System Implementation

The system was implemented with real work (not a simulation) in the laboratory of the College of Information Technology at the University of Babylon, where six laptops were used that were connected to each other by a switching device.

4.2.2 Work Environment Preparation Steps

The first step is to install Docker on all the laptops (in this thesis, six laptops will be used for the implementation) and set up a cluster configuration via Docker Swarm. The second step is to spread the service inside the cluster using the manager that was created by Docker Swarm.

4.2.3 Environment And Software install

In the proposed system, Ubuntu 20.04 has been used as an operating system (OS). Docker 1.13 has been downloaded and installed, after installing and running Docker you can download the images of Nginx 1.19.6 image, Node JS 14.16.0 image, Redis image 6.2.6, and MongoDB 5.0.3 image from

hub.docker.com which needs to be downloaded. After that, install Autocannon Benchmark for speed testing and install Postman for API and response testing. All install action steps are found in the appendix. All experiments were performed based on the devices in table 4.1. And the design is shown in figure 4.1.

Table 4.1 the Software and Hardware of Each Node

Host Name	Role	Number of Core	Clock Rate of Core	Memory Total	Bitrate	Operation System	Software Main
Node1	Worker	4	2.50 GHz	4 GB	587 Mbit/sec	Ubuntu 18.04.3 LTS	Docker 1.13
Node2	Worker	4	2.60 GHz	4 GB	588 Mbit/sec	Ubuntu 18.04.3 LTS	Docker 1.13
Node3	Worker	4	2.50 GHz	4 GB	585 Mbit/sec	Ubuntu 18.04.3 LTS	Docker 1.13
Node4	Worker	4	2.50 GHz	12 GB	623 Mbit/sec	Ubuntu 18.04.3 LTS	Docker 1.13
Node5	Worker	4	2.50 GHz	4 GB	94 Mbit/sec	Ubuntu 18.04.3 LTS	Docker 1.13
Karar	Manager	8	2.20 GHz	12 GB	942 Mbit/sec	Ubuntu 20.04.2 LTS	Docker 1.13

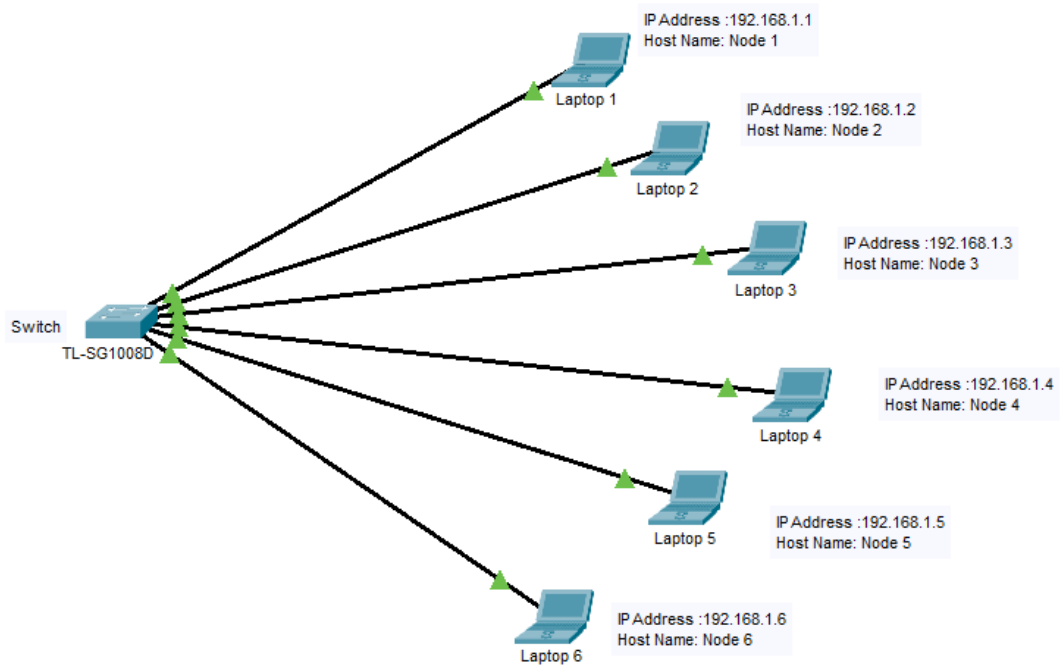


Figure 4.1 design of the connection that allows a laptops to be connected to a switch in a lab while maintaining an IP address

4.3 Docker Swarm Mode

Use appendix to create Docker Swarm mode by build a swarm on the host computer and join a node to the cluster as show in figure 4.2.

```
ubu@karar:~/Desktop/look/node-docker-main$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
rm0s1b8o52y0rtgy6okx1b4x2 *   karar      Ready     Active         Leader           20.10.8
mo7ujeg1g2mbo9dfdb6rqaqzs     node1      Ready     Active                          20.10.8
eperoq6qvkucn3du2ufd2b4rd     node2      Ready     Active                          20.10.8
rm1ylpx0kqzayr4wu3lqd59js     node3      Ready     Active                          20.10.8
q3u15mclwa8hd9vjghcic1baf     node4      Ready     Active                          20.10.8
oksvfbk986ruj39x1nga3s0c5     node5      Ready     Active                          20.10.8
```

Figure 4.2 Terminal of Docker Swarm after Adding All Nodes

4.4 Overview

In the first case of experiments, a distributed load represented by NGINX is implemented with the algorithms (Round Robin, Least Connection, Enhance Weight Round Robin and Enhance Weight Least Connection). In the second case, the results are fully optimized by using the Node JS cluster (Multi-Threading) to use the entire processor of the devices involved in processing distributed traffic.

4.5 Case Study Benchmarking Tool

Use autocannon benchmarking tool to make tests for the case study. This code used for all tests in this thesis:

Autocannon –c 450 –d 30 –l URL

Where (***-c***) is the connections: The number of connections that have been made (value of connections) and (***-d***) is duration: The length of time it took to complete the test in seconds. When the (***-l***) option is used, a third table appears that displays all of the autocannon's latency percentiles. The number of connections was chosen since it could provide a test case without any drops or errors in any requests in the cluster.

In all tests, the number of connections selected is 450 in all testers because if selected 500 number of connections, the system will drop the requests and the

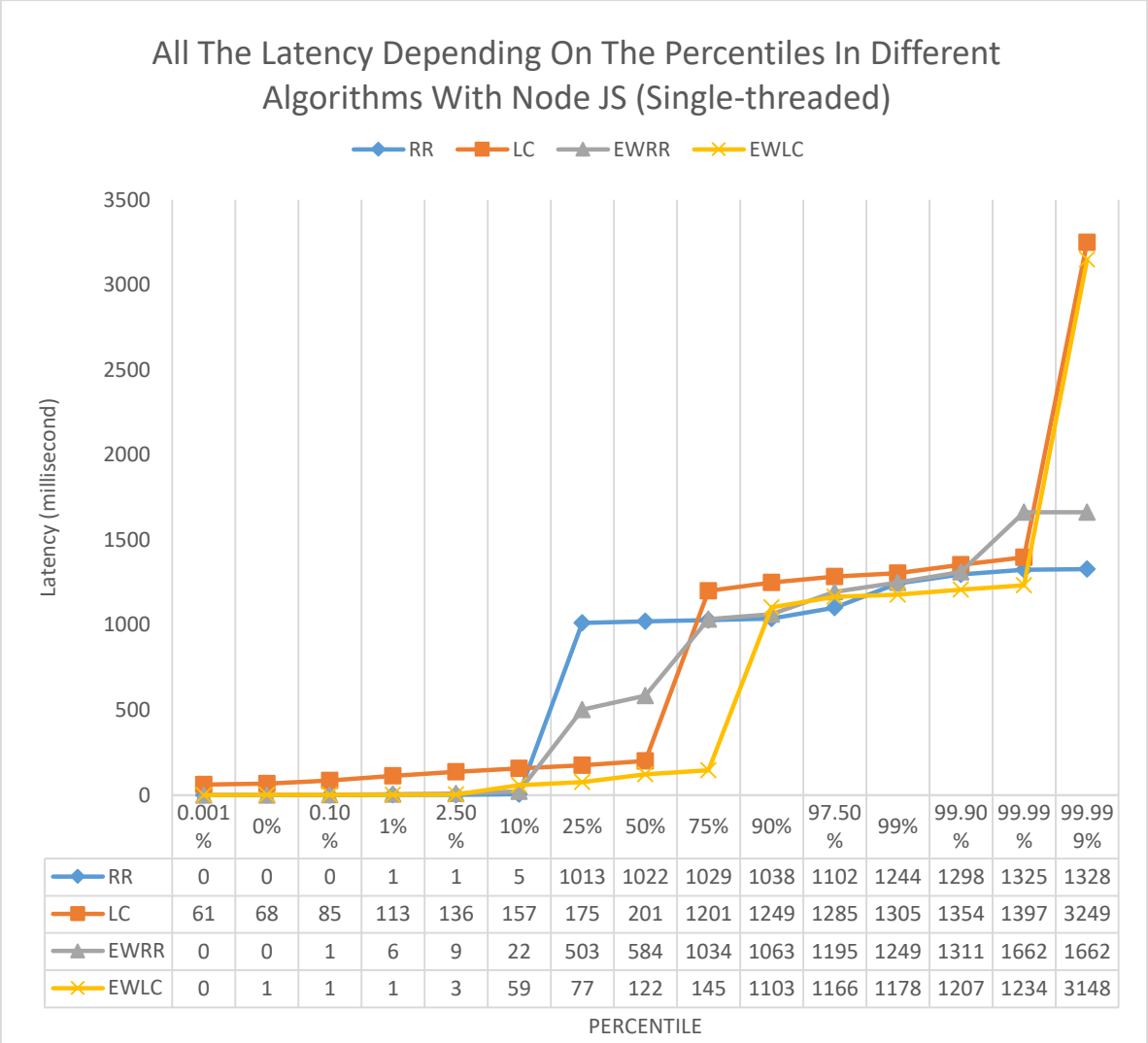
results won't show any specific difference. Because of limited hardware and network resources.

To show the result of tests of the proposed system, the time tests first use the default value of 10 seconds. However, in value, it does not show any difference because it is a fast speed to test and does not load that effect as a result. For this reason, make all the tests 30 seconds to show the result more clearly.

4.6 First Case Study Of Proposed System Enhance Algorithms

In the first case study between default algorithms and proposal algorithms, these experiments were implemented using Nginx as load balancing with the default algorithms (Round Robin, Least Connection) and proposed algorithms (Enhance Weight Round Robin, Enhance Weight Least Connection) and using the Node JS JavaScript programming language within the (Single-threaded) to build web applications.

Figure 4.3 shows the difference between the algorithms (Round Robin and Least Connection) and the proposed enhanced algorithms (Enhance Weight Round Robin and Enhance Weight Least Connection), as it fully shows all the latency depending on the percentiles.



	0.001 %	0%	0.10 %	1%	2.50 %	10%	25%	50%	75%	90%	97.50 %	99%	99.90 %	99.99 %	99.99 9%
RR	0	0	0	1	1	5	1013	1022	1029	1038	1102	1244	1298	1325	1328
LC	61	68	85	113	136	157	175	201	1201	1249	1285	1305	1354	1397	3249
EWRR	0	0	1	6	9	22	503	584	1034	1063	1195	1249	1311	1662	1662
EWLC	0	1	1	1	3	59	77	122	145	1103	1166	1178	1207	1234	3148

Figure 4.3 all the latency depending on the percentiles in different algorithms with node js (single-threaded)

The above results in figure 4.3 show that the proposed algorithm of Enhance Weight Least Connection is better than other algorithms at 25%, 50%, 75%, 99.90%, and 99.99%. Where the lower number is better, and it is measured in milliseconds. But in the percentage of 99.999%, there is a big difference. All these results of algorithms will be improved in the second case study.

The average, standard deviation, and maximum latency of requests are displayed in this statistic in the figure 4.4.

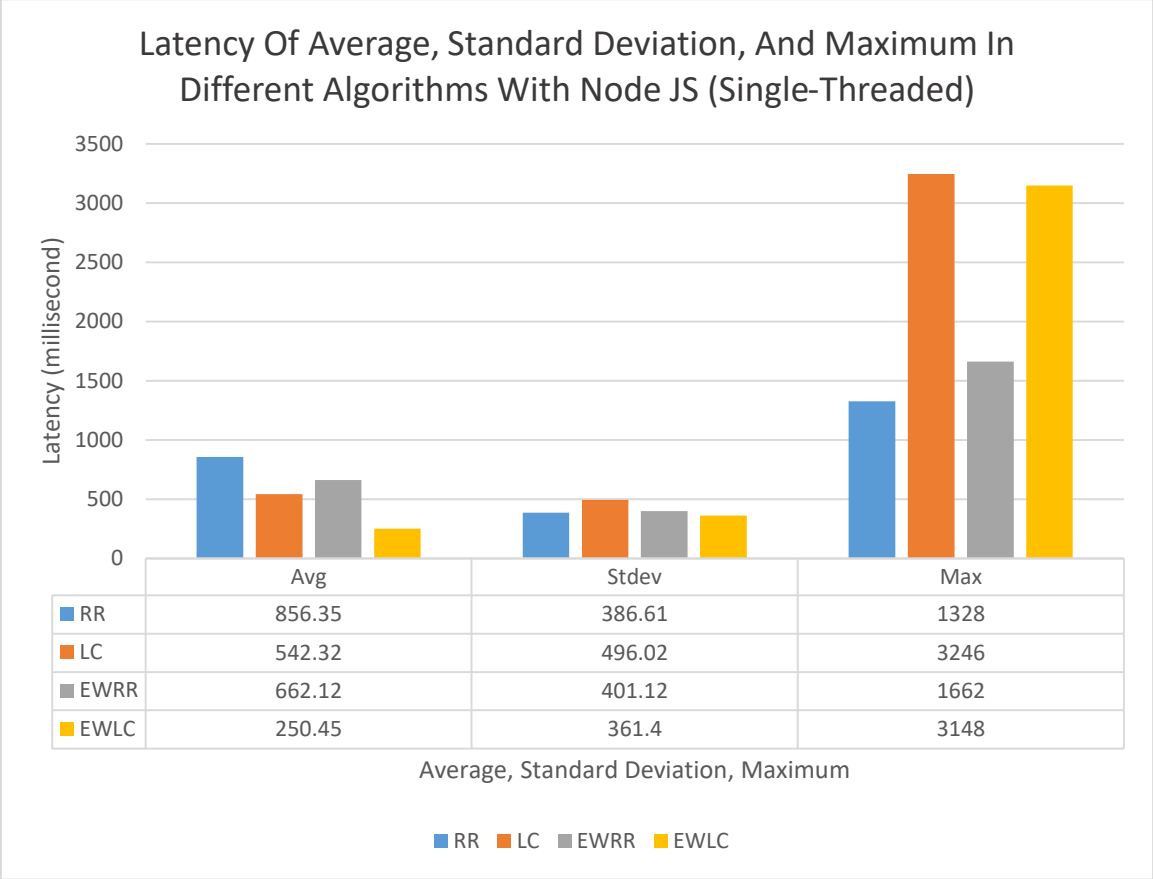


Figure 4.4 latency of average, standard deviation, and maximum in different algorithms with node js (single-threaded)

From the above results in figure 4.4, see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result in latency, where the average is equal to 250.45 and the standard deviation is equal to 361.4, where the lower the number, the better it is, and it is measured in milliseconds, and the maximum is equal to 3148. The result of maximum performance is Enhance Weight Least Connection, which is not the best because some request cases are still waiting to be executed, which is the reason for the higher maximum result of Enhance Weight Least Connection.

Here the results of the rate of the number of requests processed are displayed based on the percentage are shown in figure 4.5.

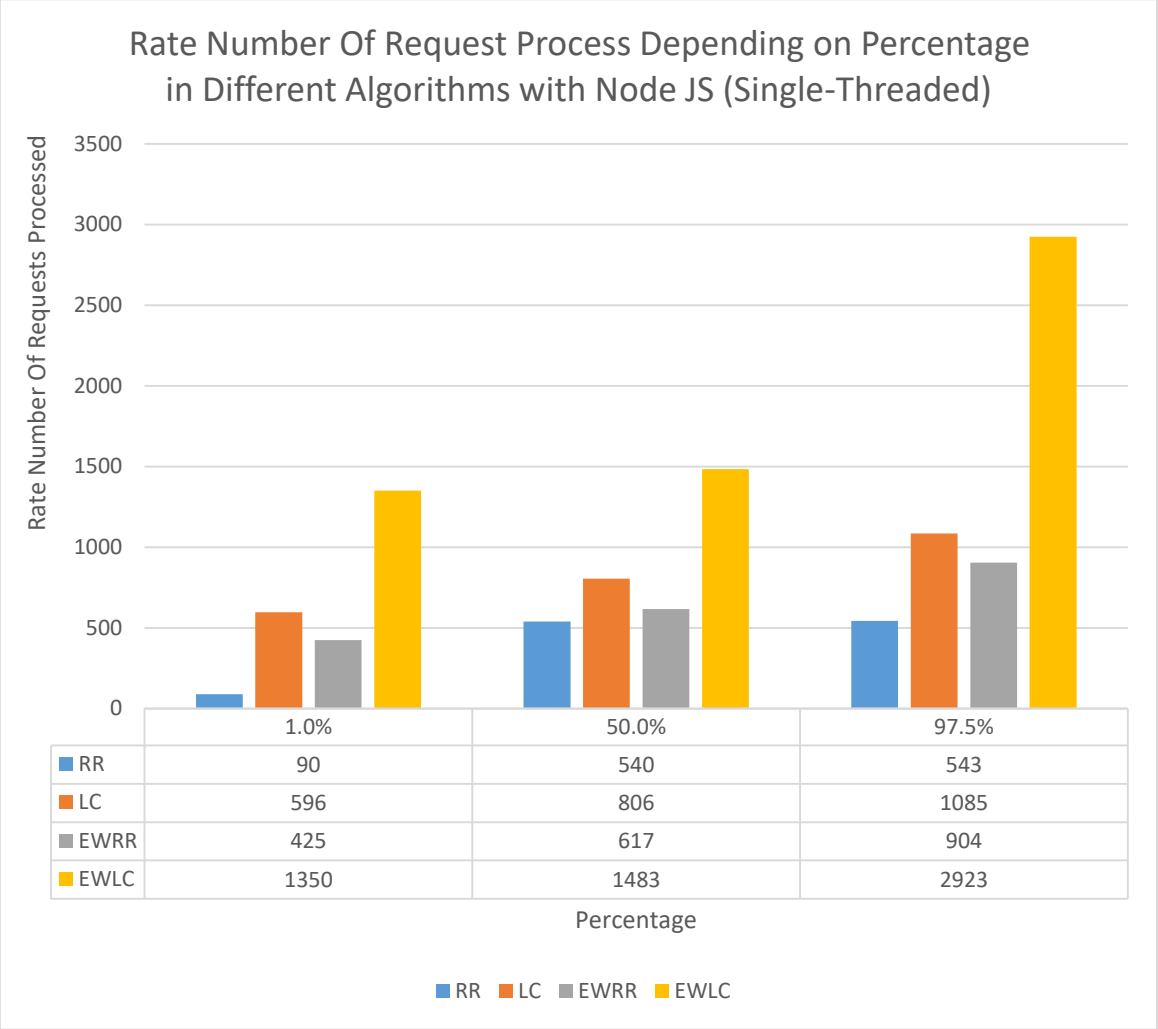


Figure 4.5 rate number of request process depending on percentage in different algorithms with node js (single-threaded)

From the apparent higher results in figure 4.5, we see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result of the requests. Where the requests when the load is 1% equal to 1350, and where 50% is equal to 1483, and where 97.5% is equal to 2923, where the higher the number, the better it is.

This figure 4.6 shows the average, standard deviation, and finally the minimum number of requests.

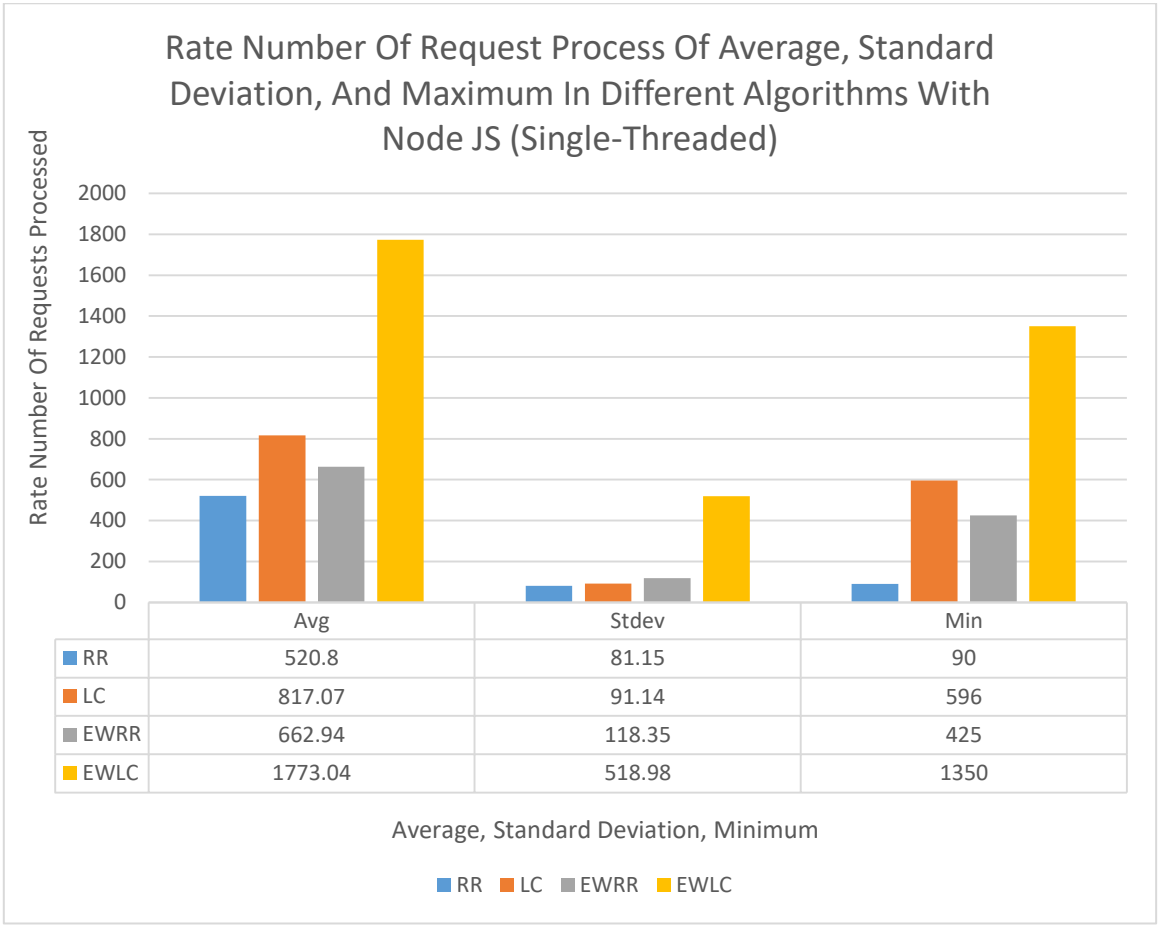


Figure 4.6 rate number of request process of average, standard deviation, and maximum in different algorithms with node js (single-threaded)

From the apparent higher results in figure 4.6, see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result in request numbers where the average is equal to 1773.04 and the standard deviation is equal to 518.98. Where the height of the number is better, and the minimum is equal to 1350 requests starting at 1% in figure 4.7.

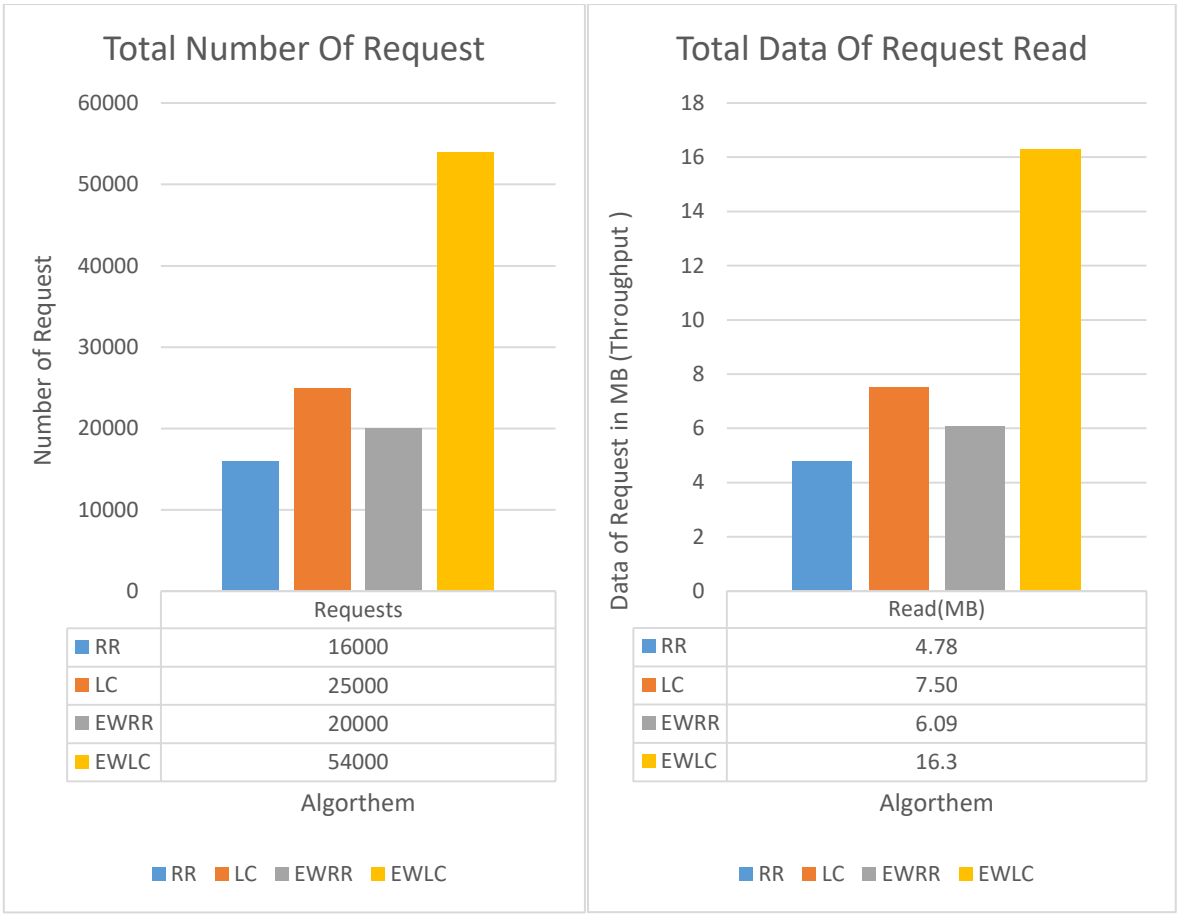


Figure 4.7 total number of request & total data of request read in MB in different algorithms with node js (single-threaded)

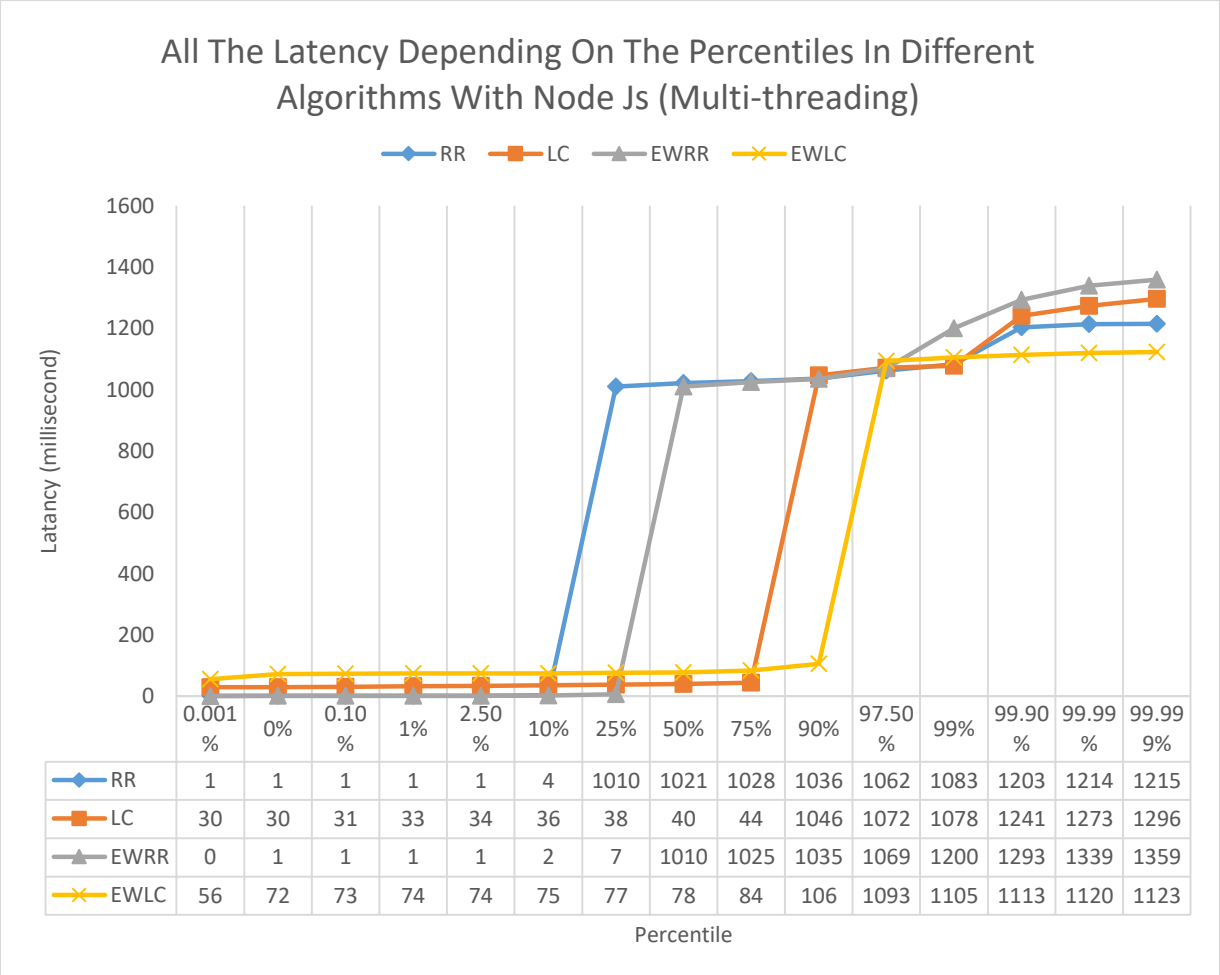
The left side of figure 4.7 shows the total number of requests for the different algorithms. The results of the Enhance Weight Least Connection show processing the highest number of requests from other algorithms.

The right side of figure 4.7 shows the throughput of different algorithms. The throughput of the Enhance Weight Least Connection shows the best record from other algorithms.

4.7 Second Case Study Of Proposed System Improve Processing

In the second case study, after the addition of cluster node JS as multi-core on each web server node of a swarm, these experiments were implemented using the first case within the cluster Node JS and included a (Multi-Threading) on each node service in a docker swarm.

This statistic in figure 4.8 shows the latency depending on the percentile difference after making a cluster (Multi-Threading) and using the algorithms (Round Robin and Least Connection) with the proposed enhanced algorithms (Enhance Weight Round Robin and Enhance Weight Least Connection).



	0.001 %	0%	0.10 %	1%	2.50 %	10%	25%	50%	75%	90%	97.50 %	99%	99.90 %	99.99 %	99.99 9%
RR	1	1	1	1	1	4	1010	1021	1028	1036	1062	1083	1203	1214	1215
LC	30	30	31	33	34	36	38	40	44	1046	1072	1078	1241	1273	1296
EWRR	0	1	1	1	1	2	7	1010	1025	1035	1069	1200	1293	1339	1359
EWLC	56	72	73	74	74	75	77	78	84	106	1093	1105	1113	1120	1123

Figure 4.8 all the latency depending on the percentiles in different algorithms with node js (multi-threading)

The above results in figure 4.8 show that the proposed algorithm of Enhance Weight Least Connection is better than other algorithms. Where the (RR at 25% and LC at 90% and EWRR at 50%) have a large difference from the previous percentage, but the EWLC has more power so, performance is low at 97.50%. But in this scenario, results are derived from a single request, which may or may not be enough. To verify the algorithm's state, look at all of the requests after putting them together in figure 4.9.

The average, standard deviation, and maximum latency of requests are displayed in this statistic in figure 4.9.

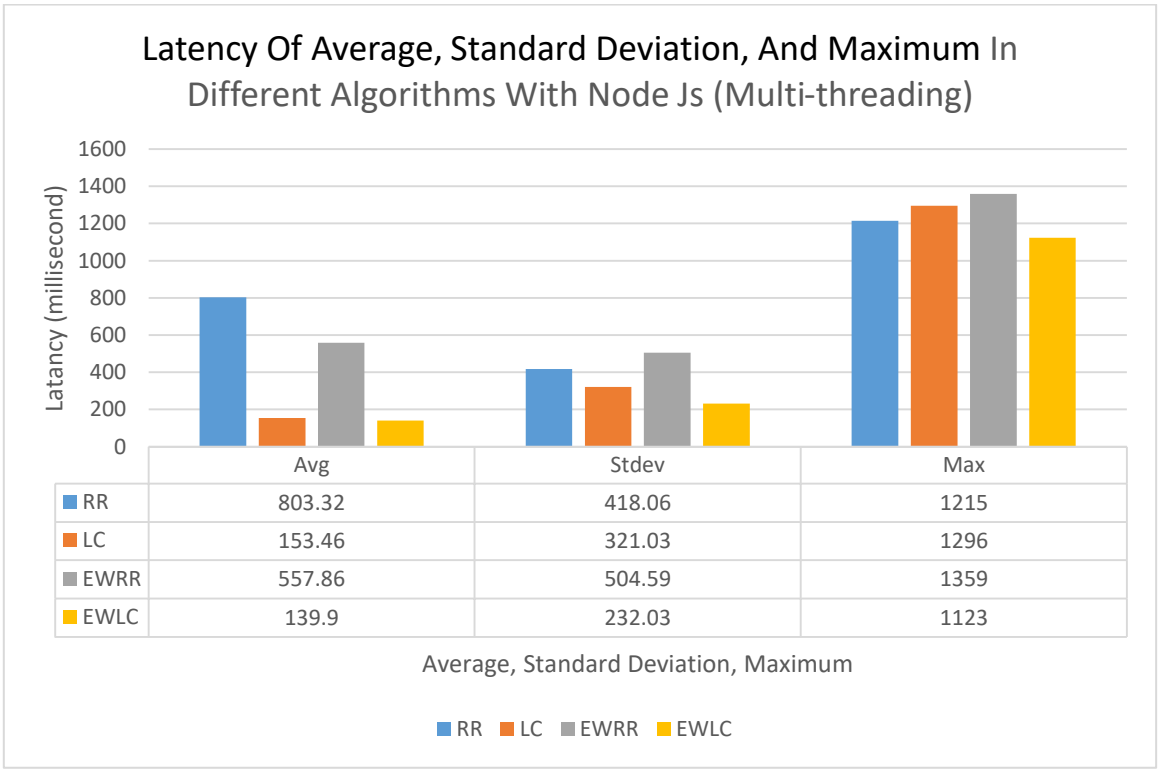


Figure 4.9 latency of average, standard deviation, and maximum in different algorithms with node js (multi-threading)

From the above results figure 4.9, see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result of latency where the average is equal to 139.9 and where is standard deviation is equal to 232.03, where the lower the number the better it is, and it is measured in milliseconds, and the maximum is equal to 1123. The result of the maximum

Enhance Weight Least Connection is the best of the other results because of the processing of the request in concurrency by threads core “cluster”.

Here in figure 4.10 the results as the rate number of requests processed are displayed based on the percentage are shown.

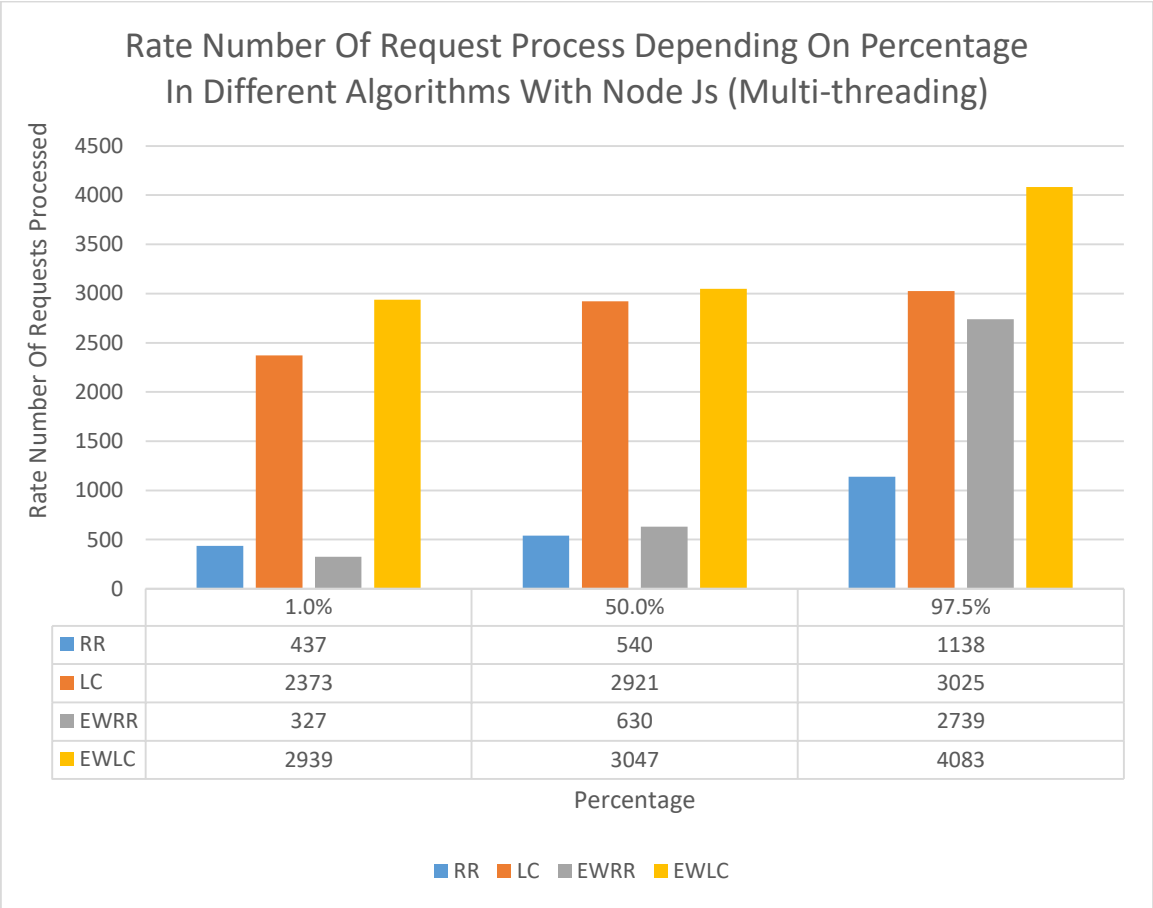


Figure 4.10 rate number of request process depending on percentage in different algorithms with node js (multi-threading)

From the apparent figure 4.10 higher results, see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result of the requests processed dependent on percentage load. Where the requests when the load is 1% equal to 2939, and where 50% is equal to 3047, and where 97.5% is equal to 4083, where the higher the number is the better it is.

This figure 4.11 shows the average, standard deviation, and finally the minimum of requests.

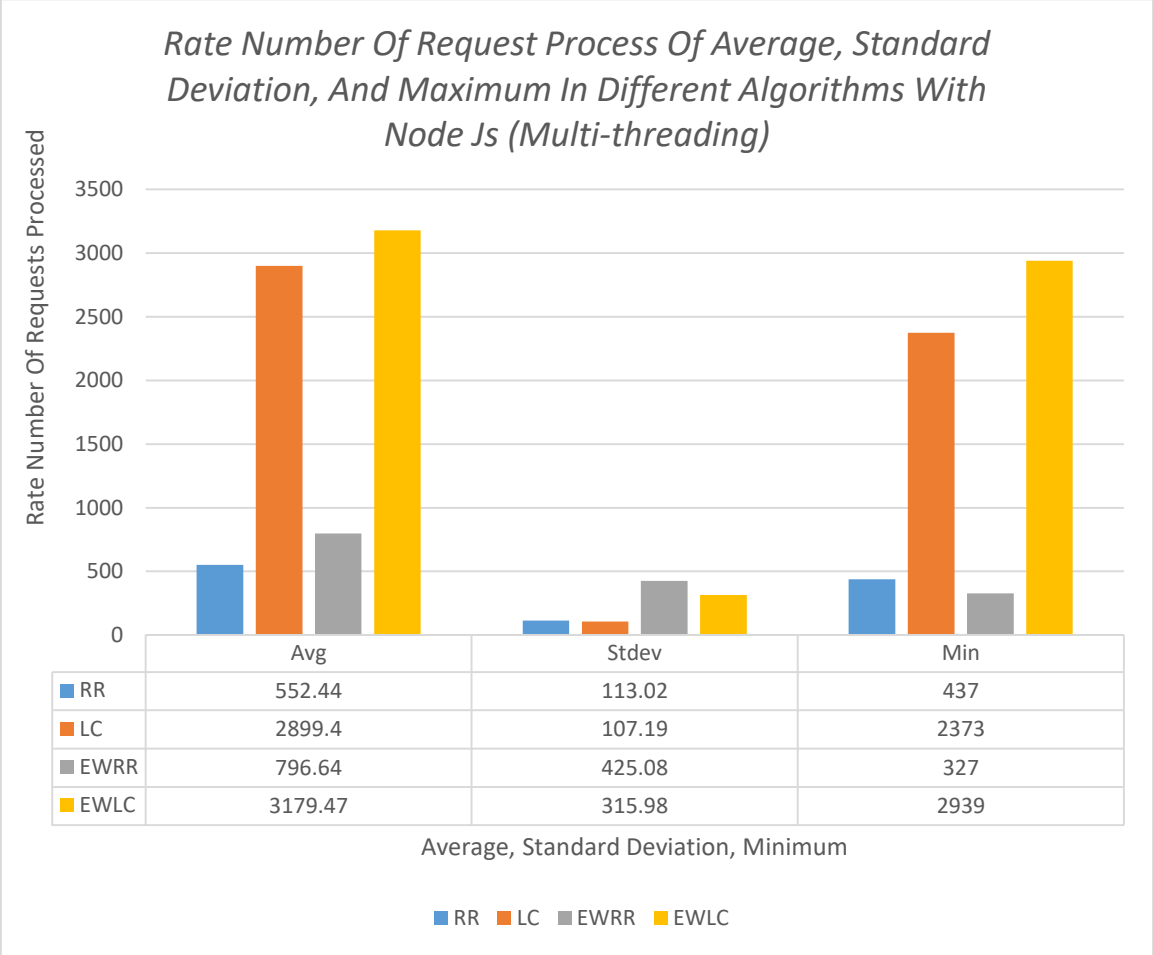


Figure 4.11 rate number of request process of average, standard deviation, and maximum in different algorithms with node js (multi-threading)

From the apparent higher results in figure 4.11, see that the performance of the Enhance Weight Least Connection is the best algorithm, as it has the best result of request number where the average is equal to 3179.47 and the is standard deviation is equal to 315.98, where the height the number the better it is, and it is measured in milliseconds, and the minimum is equal to 2939 request to start at 1%.

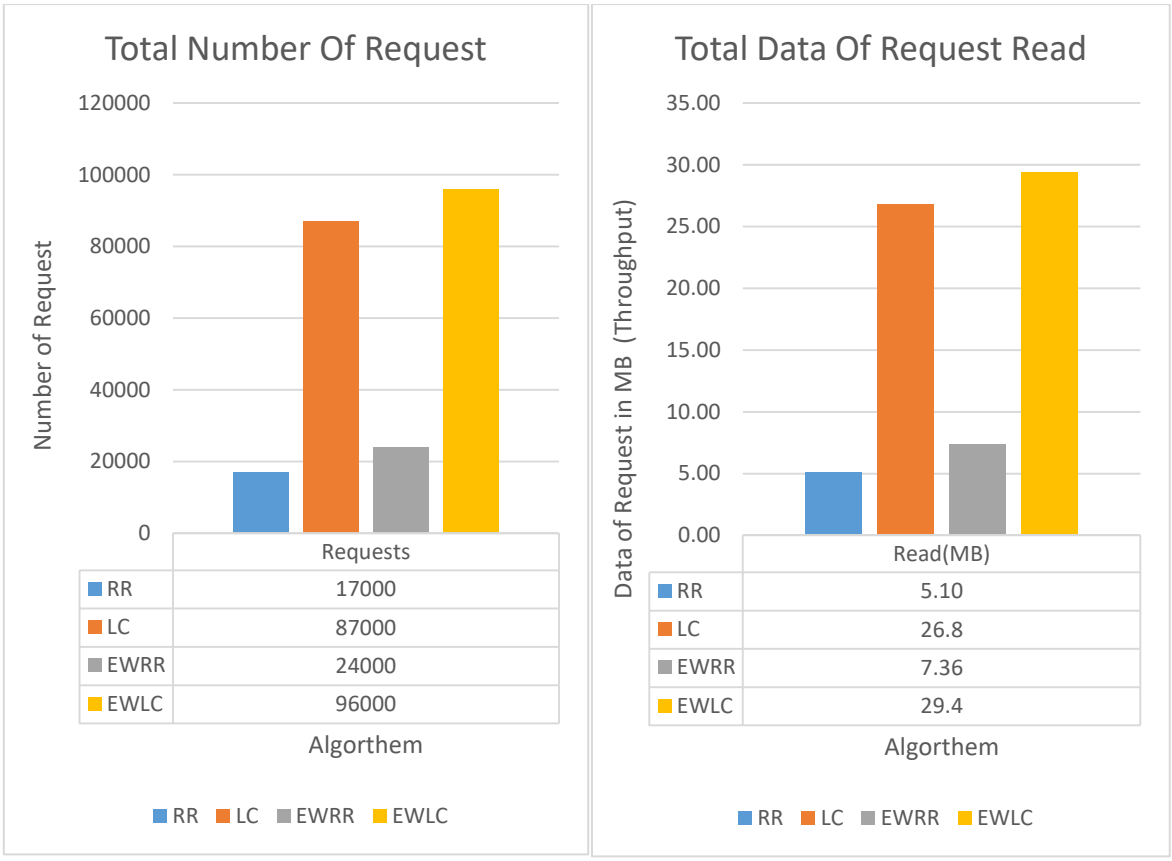


Figure 4.12 total number of request & total data of request read in MB in different algorithms with node js (multi-threading)

The left side of figure 4.12 shows the total number of requests in each of the different algorithms. The results of the Enhance Weight Least Connection show processing the highest number of requests from other algorithms.

The right side of figure 4.12 depicts the throughput of various algorithms. The throughput of the Enhance Weight Least Connection shows the best record of other algorithms.

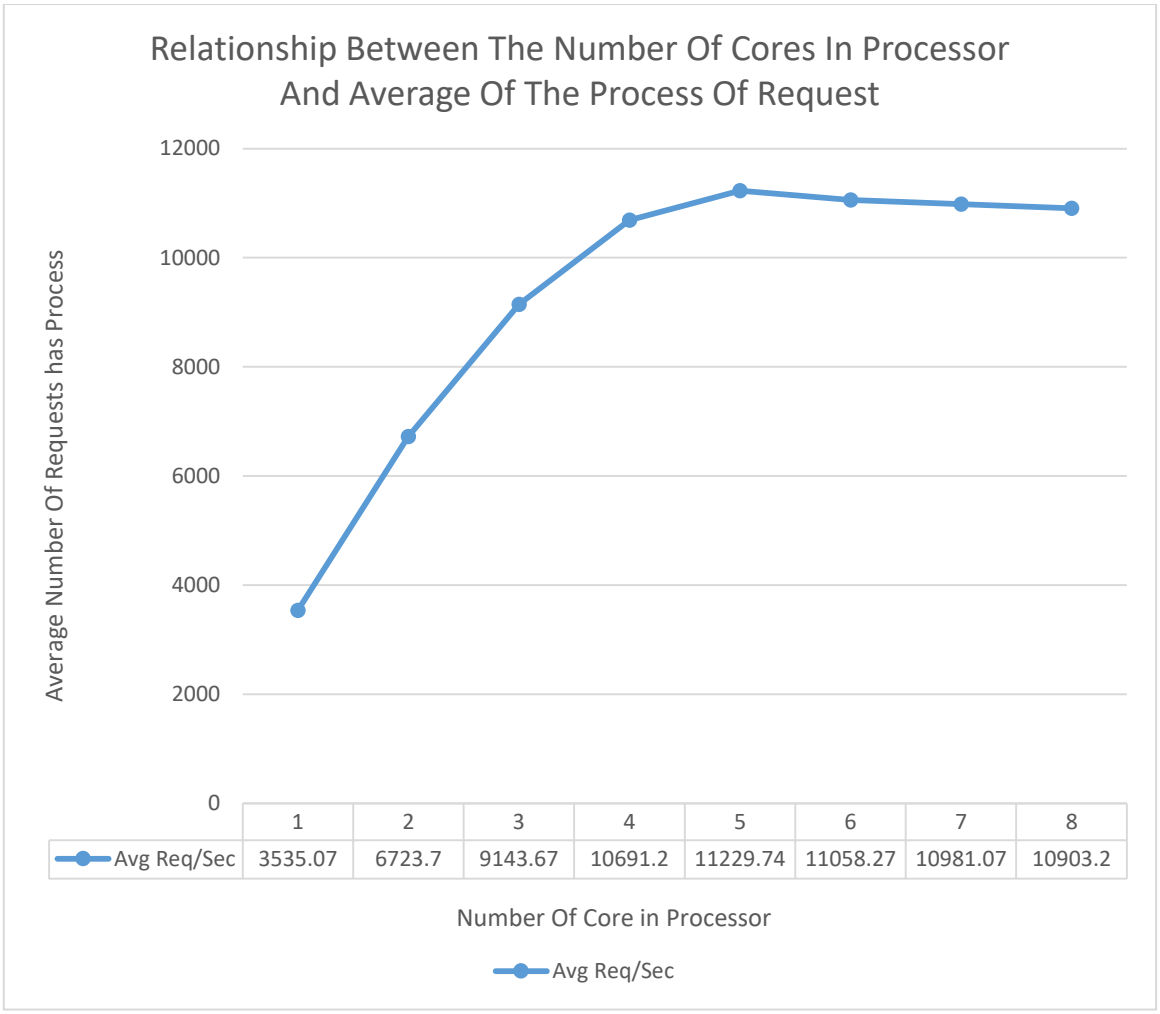


Figure 4.13 relationship between the number of cores in processor and average of the process of request

Figure 4.13 shows that the higher the number of cores in the processor machine, the faster the Web request service. When the number of cores is one, then while the weight of this machine is one, it has an average processed 3535.07 requests, while at two, it has an average processed 6723.7 requests, and at three, it has an average processed 9143.67 requests. At the end, when the number of cores is eight, the average number of requests processed is equal to 10903.2 requests. Because the system includes a total of eight cores and the operating system requires cores to run, the average of 6, 7, and 8 will not increase on a single computer.

4.8 Redis Cash Testing

The cache's ability to retrieve data such as URL links, data values, file storage, and sessions will be tested in this portion of the figure 4.14 below.

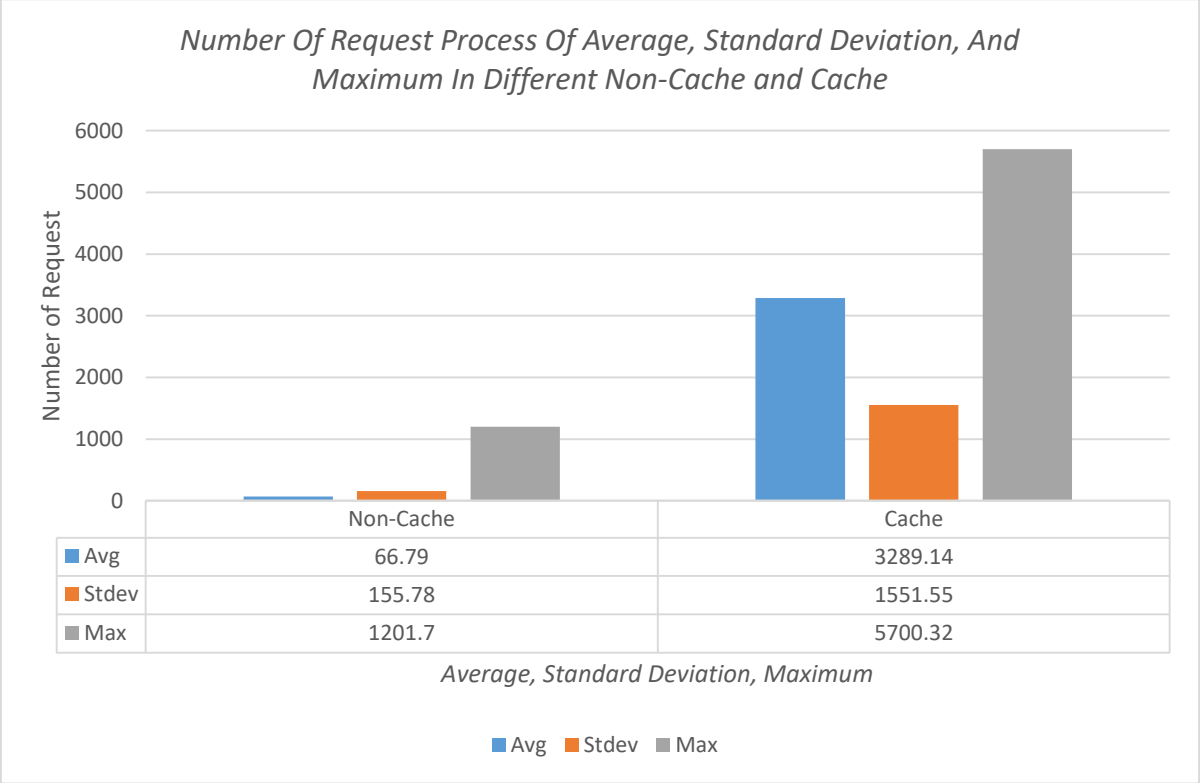


Figure 4.14 number of request process of average, standard deviation, and maximum in different non-cache and cache

Figure 4.14 shows the difference between the non-cache and cache where the average number of requests processed per second is 66.79 at the non-cache while it is 3289.14 at the cache. And the maximum number of requests that have been processed in a second is 1201.7 at non-cache and 5700.32 at cache. The standard deviation at non-cache is 155.75 and at cache is 1551.55, indicating that the number of requests in the cache is not close to the other, but is distributed in a wide range of values, indicating that the cache is superior. This test is done on a local page. They have some information, and the test is done by an autocannon benchmark tool.

In Figure 4.15, show the difference between latency in non-cache and cache. Latency is best in low values because it represents the time spent waiting for a request to be served.

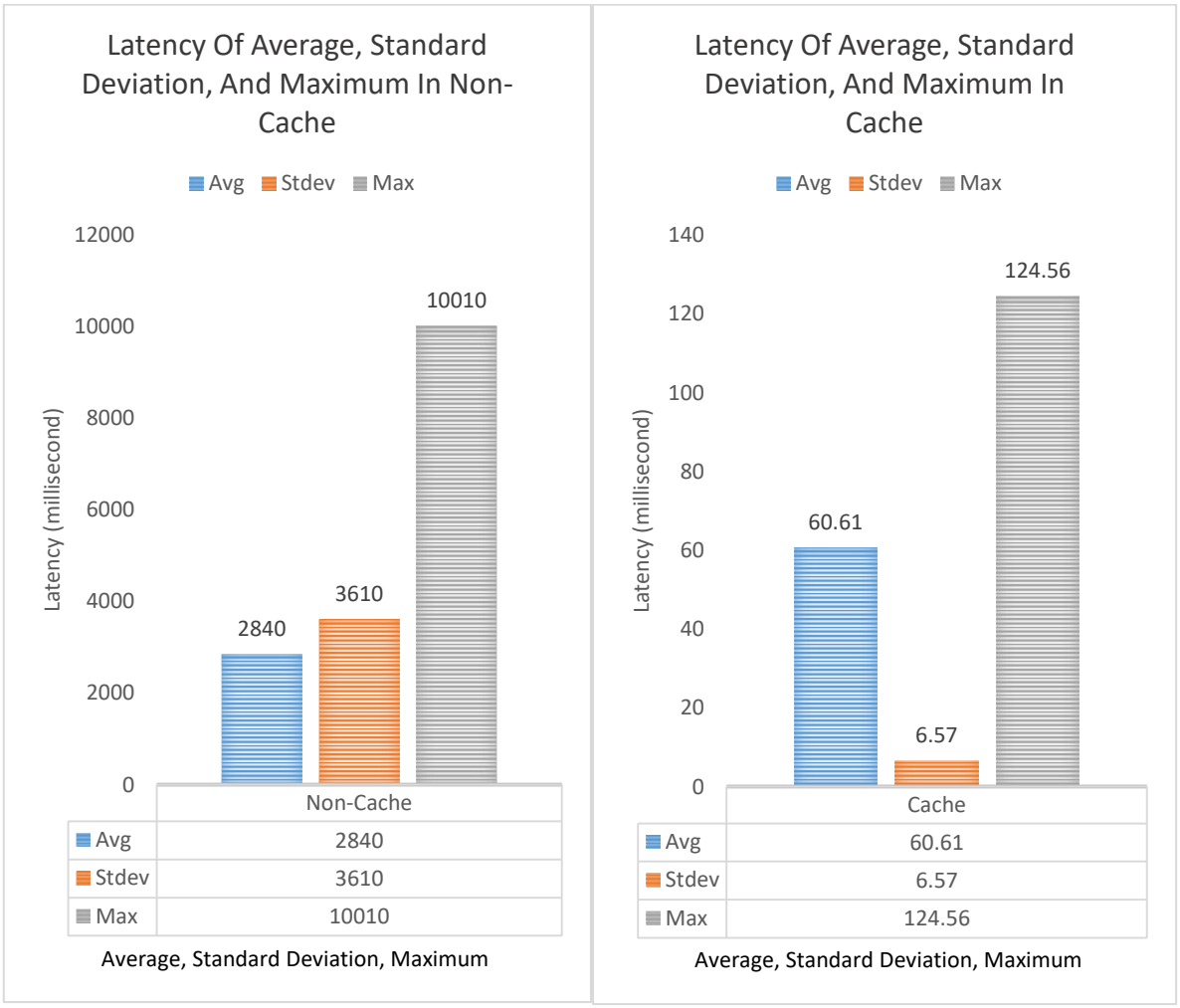


Figure 4.15 different of non-cache and cache in latency of average, standard deviation, and maximum

4.9 MongoDB GridFS Testing

Create a simple test for a storage file video using Node JS, the API, and MongoDB. The test involves uploading a file and showing how it is stored in MongoDB. File storage by GridFS stores data as binary hexadecimal in many chunks using the GridFS function. As you can see from table 4.2 below, this test uses video files because it has more size and users need some time to watch some parts of this video.

Table 4.2 GridFS in MongoDB was used to divide a video file into chunks

Normal File Storage	
File Name	bigbuck1.mp4
Length (Total Size)	454651789 kilobyte, 443995.8877 megabyte
After Use GridFS	
Chunk Size	261120 kilobyte, 255 megabyte
Number of Chunk	1742

To convert 261120 Kilobytes (KB) to Megabytes (MB), use the calculator below (MB). 255.0 Megabyte is equivalent to 261120 Kilobyte. 261120 KB to MB conversion formula is 261120 / 1024.

- Length: The size of the document in bytes.
- Chunk Size: The number of bytes that each chunk contains. In everything except the final chunk, Grid FS breaks the document into chunks of the size chunkSize; the last chunk, however, is only as big as is required. The default file size is 255 Megabytes (MB).

In Figure 4.16, the file name bigbuck1 is a video file with an mp4 format uploaded to storage in a database of videos.fs.files.

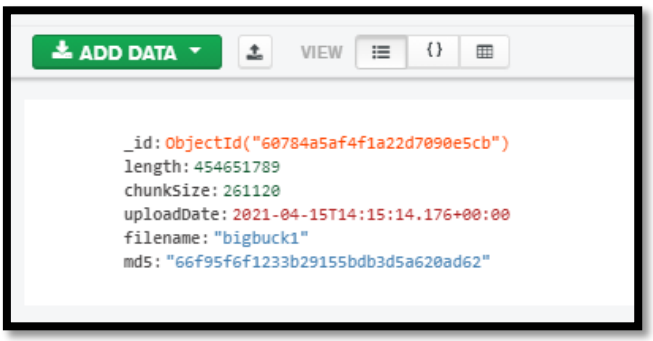


Figure 4.16 File storage of video in MongoDB in the database videos.fs.files

In Figure 4.17, after uploading video file bigbuck1, the file is divided into chunks. Each one has the _id and files_id of the file video in 4.16 and n that represent the order in chunks and lastly the data.

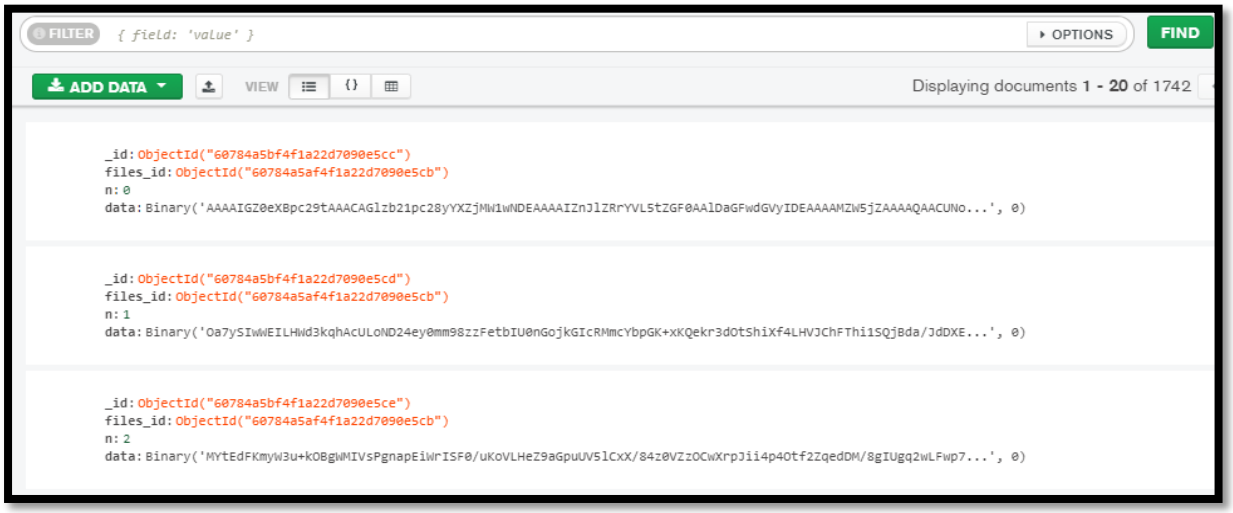


Figure 4.17 File storage of video in MongoDB in the database videos.fs.chunks

This figure 4.18 is the test after storing the file video in the normal way. The browser gets the full video as one piece, which uses more bandwidth and makes the network flow slower and places more load on the browser. This test appears in the browser after the user inspects an element in the section network, and the file size of 952 MB represents the full size of the video.

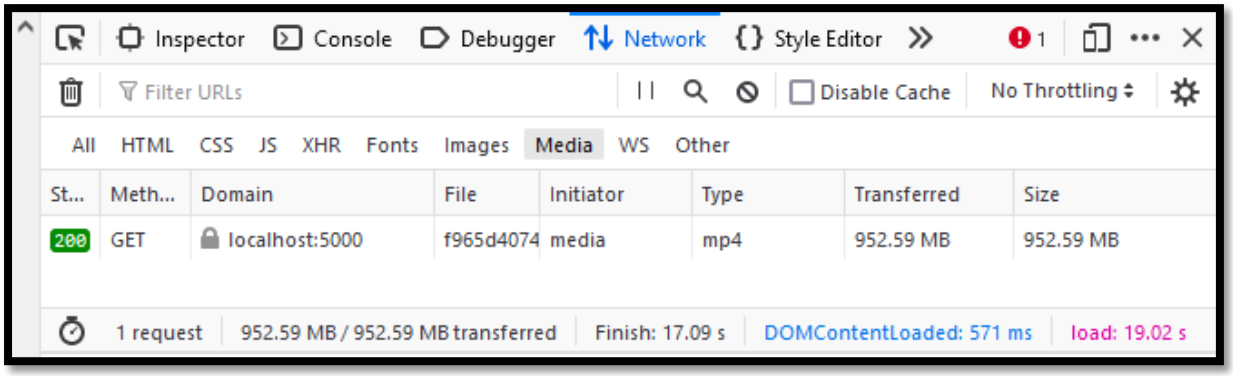


Figure 4.18 The media load with the state of regular storage video as one file is shown on the screen of the browser's inspect element.

This figure 4.19 is the test after storing the file video by using the GridFS method in MongoDB and Node JS. The browser gets the video as a chunk piece depending on the request of the client into video range time in any part of the video from the beginning to the end, which uses low bandwidth and makes the network flow faster and places a low load on the browser. This test shows up in the browser after the user looks at an element in the section network. The file size of the chunk depends on the resolution of the video and changes as the video plays.

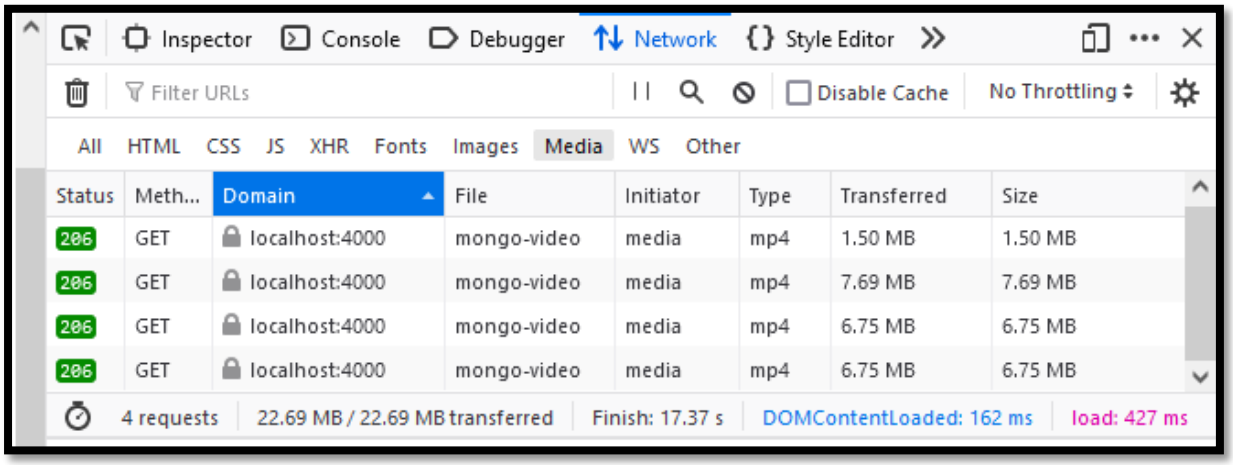


Figure 4.19 The media load is shown the status of Grid FS storage video on the inspect element of the browser depending on the request of the client on any part of the video

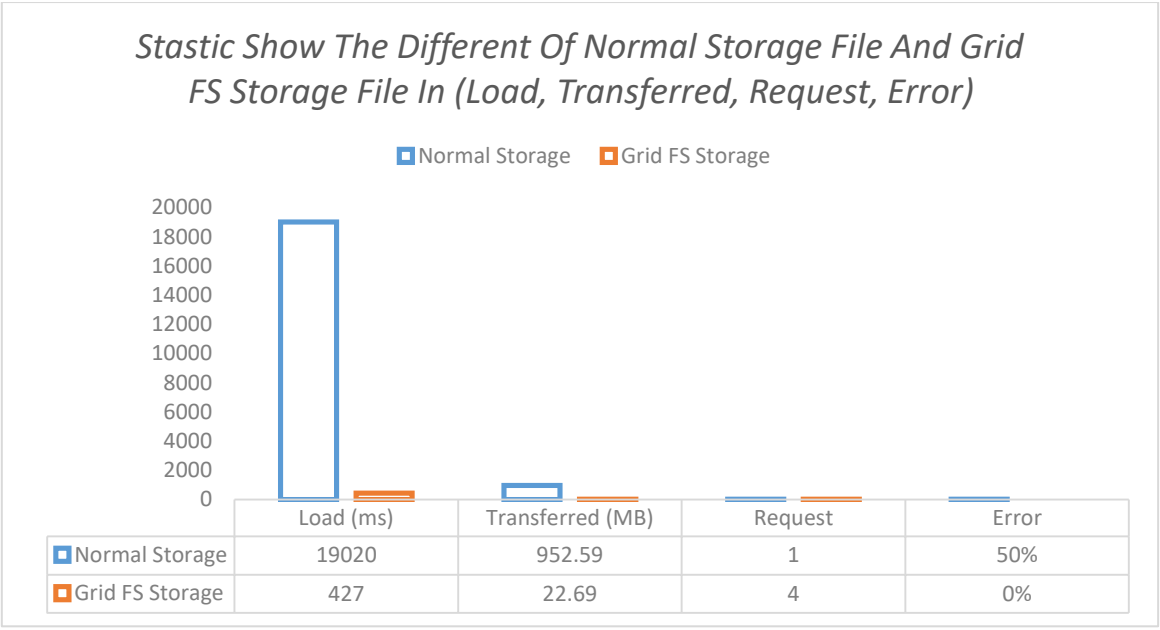


Figure 4.20 The difference between a standard storage file and a Grid FS storage file in (Load, Transferred, Request, Error)

- Load: represents the time in milliseconds to connect to the download file media.
- Transferred: represents the size of the file transfer on network media.
- Request: The number of requests made by media users is represented.
- Error: An error in media represents a blunder, such as a slow, crash, or error sound.

As a summary and explain figure 4.20, MongoDB Grid FS is better because it divides files into chunks. While in normal storage, this causes more RAM and more bandwidth network traffic, which slows down the system and makes the database server more work.

4.10 System Functionality Testing

System functionality testing is done by testing the failover mechanism, which is a mechanism that exists in the docker swarm. This test is done in case one of the hosts of the swarm cluster dies, or By turning off one of the hosts, as shown in figure 4.21.

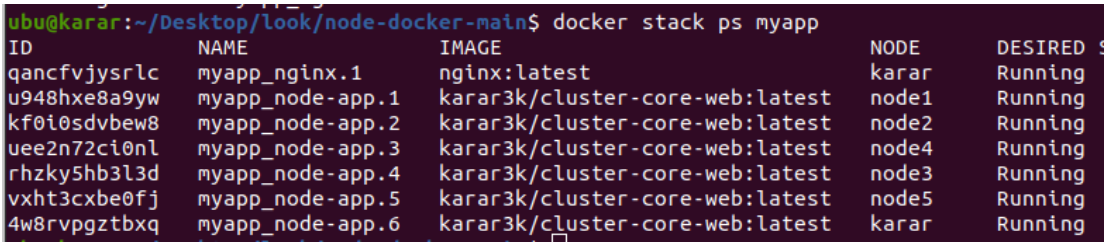


Figure 4.21 Location of the initial container

Figure 4.22 shows the location of the container after one worker2 host is turned off. It can be seen that the backend service that was previously on host worker2 moved to host worker1. This shows that the failover mechanism works well.

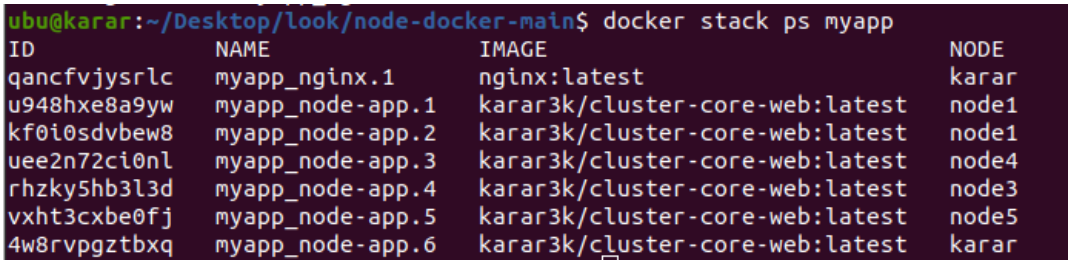


Figure 4.22 Place the final container

Chapter Five

Conclusions And Future Works

Chapter Five

Conclusions And Future Works

5.1 Conclusions

There are several conclusions that can be drawn from the thesis' recommended methodologies, which may be summarized as follows:

1. The failover mechanism contained in the docker swarm also works well, so that the system has a high level of availability.
2. The fault tolerance, defined as an algorithm's capacity to provide uniform load balancing in the event of a node or link failure, also works well. As a consequence of this, load balancing is able to be a mechanism that can tolerate errors.
3. The designed system can increase availability because it is able to failover when a failure occurs either on the load balancer side or on the backend server-side.
4. The use of the failover and the fault tolerance allows the system to stabilize and maintain system performance, which can be used as a solution for high-traffic sites in order to ensure that there are no issues with the system and that web apps can be deployed on it.
5. The proposed system's throughput results with (RR) is 5.10 MB, (LC) is 26.80 MB, (EWRR) is 7.36 MB, and (EWLC) is 29.40 MB. (EWLC) has the best-recorded throughput.
6. The proposed system's latency average with (RR) is 803 ms, (LC) is 153 ms, (EWRR) is 557 ms, and (EWLC) is 139 ms. (EWLC) has the best-recorded latency.
7. The proposed system's average response time with (RR) is 552 Req/Sec, (LC) is 2899 Req/Sec, (EWRR) is 796 Req/Sec, and (EWLC) is 3179

Req/Sec. (EWLC) gives the best average response time ever recorded compared to other methods.

8. Readers should be able to use a cluster module to run a Node JS application on several CPU cores after reading this thesis. Having this level of experience will allow readers to more effectively manage and expand the company app business.
9. The benefit of using Redis with load balancing on the web is that it makes the session cache and stores data as key-value in RAM and returns faster. It also uses Redis because of scenarios (no buffer cache) where client requests are sent to two or more replicated web servers through a load balancing method.
10. The benefit of using MongoDB with load balancing on the web is that it makes the database side faster because it can use two or more databases so that when executing requests that require heavy queries, many requests are served fast. MongoDB support for video streaming captures video data and sends it to the client in tiny chunks.

5.2 Thesis Limitations

Here are a few points that highlight the thesis's limits.

1. Malicious requests will not be discussed in any way.
2. The lab's hardware (Clock Rate "CPU Speed in GHz") at the University of Babylon's College of Information Technology limits maximum performance.

5.3 Future Works

Some ideas will be suggested as future works as explained below:

1. Applying the proposed method over different parallel platforms such as grid or cloud or both.

2. Understanding the effect of various factors on load balancing such as Clock Rate (CPU Speed in GHz).
3. The balancer can be divided into two pieces, and one of those components serves as a backup balancer. In order to improve the availability of the servers.
4. In the future, it would be interesting to compare the proposed that was suggested with other ways that are used to solve other types of issues, such as load balancing with GPU processing.
5. Applying this thesis to artificial intelligence (AI) and machine learning (ML) using tools like TensorFlow.js platform.

References

[1] A. E. Borko Furht, Handbook of Cloud Computing, Springer New York Dordrecht Heidelberg London: Springer Science & Business Media, 2010.

[2] D. H. E. P. Peter Membrey, Practical Load Balancing, Springer Science & Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013: Paul Manning, 2012.

[3] N. Naik, "Migrating from Virtualization to Dockerization in the Cloud: Simulation and Evaluation of Distributed Systems," *2016 IEEE 10th International Symposium on the Maintenance and Evolution of Service-Oriented and Cloud-Based Environments (MESOCA),* pp. 1-8, 2016.

[4] N. Naik, "Building a Virtual System of Systems Using Docker Swarm in Multiple Clouds," *2016 IEEE International Symposium on Systems Engineering (ISSE),* p. 7753148, 2016.

[5] T. T. Abdul Aziz, "Analisis Web Server untuk Pengembangan Hosting Server Institusi: Pembandingan Kinerja Web Server Apache dengan Nginx (Web Server Analysis for Institutional Server Hosting Development: Comparing Apache Web Server Performance with Nginx)," *Journal Multinetics,* vol. 1, pp. 12-20, 2015.

[6] M. S. Poonam Kumari, "A Round-Robin Based Load Balancing Approach for Scalable Demands and Maximized Resource Availability," *International Journal of Engineering and Computer Science,* vol. 5, no. 8, pp. 17375-17380, 2016.

[7] F. M. Alam Rahmatulloh, "Implementasi Load Balancing Web Server Menggunakan Haproxy Dan Sinkronisasi File Pada Sistem Informasi Akademik Universitas Siliwangi (Implementation of Load Balancing Web Server Using Haproxy and File Synchronization in the Academic Information System)," *Jurnal Nasional Teknologi Dan Sistem Informasi,* vol. 3, p. 241–248, 2017.

[8] Z. E. Intan Ferina Irza, "Analisis Perbandingan Kinerja Web Server Apache dan Nginx Menggunakan Httperf Pada Portal Berita (Studi Kasus beritalinux.com) (Comparative Analysis of Apache and Nginx Web Server Performance Using Httperf on News Portals (Case Study of beritalinux.com))," *Jurnal Vokasional Teknik Elektronika & Informatika (Electronic & Informatics Engineering Vocational Journal),* vol. 5, p. 2, 2017.

[9] Y. L. W. L. Ruoyu Li, "An Integrated Load-balancing Scheduling Algorithm for Nginx-Based Web Application Clusters," *Journal of Physics: Conference Series,* vol. 1060, p. 12078, 2018.

[10] K. K. Gurasis Singh, "An improved weighted least connection scheduling algorithm for load balancing in web cluster systems," *International research journal of engineering and technology (irjet),* vol. 05, no. 03, p. 1950~1955, 2018.

[11] D. Y. Y. A. F. Dimara Kusuma Hakim, "Pengujian Algoritma Load Balancing pada Web Server Menggunakan NGINX (Testing the Load Balancing Algorithm on a Web Server Using NGINX)," *Jurnal Riset Sains dan Teknologi (Journal of Science and Technology Research),* vol. 3, pp. 85-92, 2019.

[12] R. C. B. Y. G. W. Luthfan Hadi Pramono, "Round-robin Algorithm in HAProxy and Nginx Load Balancing Performance Evaluation: a Review," *researchgate.net, Stmik Akakom,* 2018.

[13] R. T. S. M. H. V. A. Kresna Adi Pratama, "Implementasi laod balancing pada web server menggunakan apache dengan server mirror data secara real time," *Jurnal digit,* vol. 11, p. 178~189, 2021.

[14] V. S. P. M. V. P. S. P. Murugesan, "Enhanced load balancing in distributed web server systems using document replication mechanism," *International journal of aquatic science,* vol. 12, no. 03, p. 1697~1704, 2021.

[15] "Docker Documentation," [Online]. Available: https://docs.docker.com/get-docker/.

[16] A. Mouat, Using Docker, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2016.

[17] "Docker Hub Container Images Library," [Online]. Available: https://hub.docker.com/.

[18] "Docker Overview in Docker Documentation," [Online]. Available: https://docs.docker.com/get-started/overview/.

[19] "Use Cases Of Docker | Docker Helps Development Teams," [Online]. Available: https://www.docker.com/use-cases.

[20] "Swarm Mode Overview," [Online]. Available: https://docs.docker.com/engine/swarm/.

[21] Y. Y. C. S. H. K. Shang Liang Chen, "CLB: A Novel Load Balancing Architecture and Algorithm for Cloud Services," *Computers & Electrical Engineering,* vol. 58, pp. 154-160, 2017.

[22] S. K. V. P.P. Geethu Gopinath, "An In-Depth Analysis and Study of Load Balancing Techniques in the Cloud Computing Environment," *Procedia Computer Science,* vol. 50, pp. 427-432, 2015.

[23] "Nginx HTTP Load Balancing," [Online]. Available: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

[24] R. H. I. S. Fanny Aulia Fadillah, "Implementasi Load Balancing Nginx Algoritma Weighted Round Robin," *researchgate.net, Siliwangi University,* 2019.

[25] D. DeJonghe, NGINX Cookbook, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2021.

[26] F. S. Rahmad Dani, "Perancangan Dan Pengujian Load Balancing Dan Failover Menggunakan Nginx(Load Balancing And Failover Planning And Testing Using Nginx)," *Jurnal Ilmu Komputer dan Informatika(Journal of Computer Science and Informatics),* vol. 3, pp. 43-50, 2017.

[27] "Nginx Documentation," [Online]. Available: https://nginx.org/en/docs/.

[28] "Node JS Documentation," [Online]. Available: https://nodejs.org/en/docs/.

[29] "Node JS Tutorial," [Online]. Available: https://www.tutorialspoint.com/nodejs/index.htm.

[30] "Node JS Cluster Documentation," [Online]. Available: https://nodejs.org/api/cluster.html.

[31] F. O. Tiago Macedo, Redis Cookbook, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2011.

[32] J. L. Carlson, Redis in Action, 20 Baldwin Road, PO Box 261, Shelter Island, NY 11964: Manning Publications Co, 2013.

[33] "Redis Documentation," [Online]. Available: https://redis.io/documentation.

[34] K. Chodorow, Scaling MongoDB, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2011.

[35] S. E. MongoDB: The Definitive Guide, MongoDB The Definitive Guide, Second Edition, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2013.

[36] P. B. S. V. D. G. T. H. Kyle Banker, MongoDB in Action, Second Edition, 20 Baldwin Road, PO Box 761, Shelter Island, NY 11964: Manning Publications Co, 2016.

[37] "MongoDB Documentation," [Online]. Available: https://docs.mongodb.com/manual/introduction/.

[38] M. Wilson, Building Node Applications with MongoDB and Backbone, United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2013.

[39] "MongoDB Tutorial," [Online]. Available: https://www.tutorialspoint.com/mongodb/index.htm.

[40] E. M. Hahn, Express in Action Writing, Building And Testing NodeJS Applications, 20 Baldwin Road, PO Box 761,Shelter Island, NY 11964: Manning Publications Co, 2016 .

[41] P. R. Harihara Subramanian, Hands-On RESTful API Design Patterns and Best Practices, Livery Place, 35 Livery Street, Birmingham, B3 2PB, UK.: Packt Publishing Ltd, 2019.

[42] S. Patni, Pro RESTful APIs Design, Build and Integrate with REST, JSON, XML and JAX-RS, Santa Clara, California, USA: Library of Congress Control Number: 2017936942, 2017.

[43] "cloudcheckr.com, Horizontal vs. Vertical Scaling in the Cloud," [Online]. Available: https://cloudcheckr.com/cloud-automation/horizontal-vertical-cloud-scaling/.

[44] W. M. S. R. A B M Moniruzzaman, " A High Availability Clusters Model Combined with Load Balancing and Shared Storage Technologies for Web Servers," *arXiv: Distributed, Parallel, and Cluster Computing,* 2014.

[45] D. S. A. P. U. C. A. N. Dea Muhamad Yasin, "Performansi Waktu Respon Nosql Mongodb Dan Implementasi Nginx Load Balancer (Performance Of Nosql Mongodb Response Time And Implementation Of Nginx Load Balancer)," *researchgate.net & Universitas Siliwangi,* 2021.

[46] "Postman Learning Center , Official Information On How To Use Postman In Your API Projects," [Online]. Available: https://learning.postman.com/docs/getting-started/introduction/.

[47] "Autocannon HTTP Benchmarking Tool," [Online]. Available: https://github.com/mcollina/autocannon.

[48] "Docker Pull an image or a repository from a registry," [Online]. Available: https://docs.docker.com/engine/reference/commandline/pull/.

[49] "Docker Push an image or a repository to a registry," [Online]. Available: https://docs.docker.com/engine/reference/commandline/push/.

[50] "NPM Package Manager For The JavaScript," [Online]. Available: https://docs.npmjs.com/downloading-and-installing-node-js-and-npm.

Letter of Acceptance

Final Acceptance Letter -paper ("Web Server Load Balancing Based on Number of Client Connections on Docker Swarm ") submitted to the [IT-ELA 2021]

البريد الوارد × خارجيون

<it-ela2021@baghdadcollege.edu.iq> IT-ELA 2021

zaid.alhaboobi ،dr_bansalman ،أنا

السبت، 8 يناير 1:34 ص

اخفاء خيارات الترجمة من الإنجليزية × ترجمة الرسالة العربية الإنجليزية

Dear Author (s),

Congratulations -your paper **("Web Server Load Balancing Based on Number of Client Connections on Docker Swarm ")** has been accepted for oral presentation. Please find in the attachments the final acceptance.

Best regards,

TPC Chair

2nd International Scientific Conference on Information Technology to Enhance e-learning and other application (IT-ELA2021).

Formal Acceptance

2nd International Conference of Information Technology to enhance E-learning and other Application-2021

[IT-ELA 2021]

To/ Karar Haider Anun, Mahdi S. Almhanna

Dear respected author(s):

With heartiest congratulations. We are pleased to inform you that based on the recommendations of the reviewers and the Technical Committees, your paper entitled:

"Web Server Load Balancing Based on Number of Client Connections on Docker Swarm"

It has been accepted for oral presentation at the 2nd International Conference of Information Technology to enhance E-learning and other application-2021[IT-ELA 2021], technically sponsored by IEEE, to be held on Dec 28-29, 2021, Baghdad, Iraq.

Your paper will be submitted (**within the conference proceeding**) to the IEEE Xplore digital library for (**final acceptance of uploading to the Digital library**).

IT-ELA 2021 Baghdad – Iraq

Baghdad College of Economic Sciences University

Computer Sciences Department

IEEE

Email:it-ela2021@baghdadcollege.edu.iq

Website: https://baghdadcollege.edu.iq/it-ela2021

ID: 1570774323

Appendix

I. Install Docker on Ubuntu 20.04

It's possible that the Docker installation package in the official Ubuntu repository isn't the latest edition. We'll install Docker from the official Docker repository to ensure we obtain the most recent version. To do so, we'll create a new package source, add the Docker GPG key to verify the downloads, and then install the package.

First, make sure your package list is up to date:

```
1  sudo apt update
```

Install the following required packages to allow apt to utilize HTTPS packages:

```
2  sudo apt install apt-transport-https ca-certificates curl software-properties-common
```

Then add the official Docker repository's GPG key to your system:

```
3  curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
```

To add the Docker repository to APT sources, do the following:

```
4  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu focal stable"
```

This will also add the Docker packages from the newly added repo to our package database. Make sure you're installing from the Docker repo rather than the Ubuntu default repo:

```
5  apt-cache policy docker-ce
```

You'll see something like this. However, the Docker version number may be different:

```
docker-ce:
  Installed: (none)
  Candidate: 5:19.03.9~3-0~ubuntu-focal
  Version table:
     5:19.03.9~3-0~ubuntu-focal 500
        500 https://download.docker.com/linux/ubuntu focal/stable amd64 Packages
```

The candidate for installation is from the Docker repository for Ubuntu 20.04, and docker-ce is not installed.

Finally, run the following commands to install Docker:

```
sudo apt install docker-ce
```

Docker should now be installed, the daemon running, and the process set to start automatically when the computer boots up. Make sure it's up and running:

```
sudo systemctl status docker
```

The output should look something like this, indicating that the service is up and running:

```
Output
● docker.service - Docker Application Container Engine
     Loaded: loaded (/lib/systemd/system/docker.service; enabled; vendor preset: enabled)
     Active: active (running) since Tue 2021-06-20 17:00:41 UTC; 17s ago
TriggeredBy: ● docker.socket
       Docs: https://docs.docker.com
   Main PID: 24321 (dockerd)
      Tasks: 8
     Memory: 46.4M
     CGroup: /system.slice/docker.service
             └─24321 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

When you install Docker, you get both the Docker service (daemon) and the docker command-line program, commonly known as the Docker client.

II. Using Docker Hub to Pull and Push Image

A- Pull an image from Docker Hub

Most of your images will be built on top of a Docker Hub base image. You can get and try numerous pre-built images from Docker Hub without having to specify and set up your own. Use docker pull to get a specific image or collection

of images. If no tag is specified, the Docker Engine defaults to the:latest tag. The "*debian:latest"* image is retrieved with this command [48].

```
1  docker pull NAME[:TAG]
```

B- Push an image to Docker Hub

To upload your images to the Docker Hub registry, use docker image push. During docker push, progress bars are displayed to represent the uncompressed size. The real amount of data pushed will be compressed before being sent, so the progress meter will not represent the uploaded size. Docker login manages registry credentials. Examples: Create a new image and save it to the registry. The new image is saved by first locating the container ID (using docker container ls) and then committing it to a new image name [49].

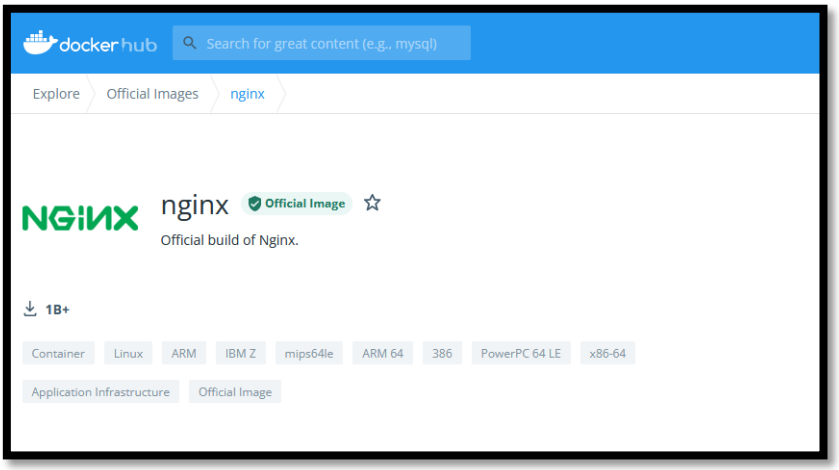
```
2  docker push NAME[:TAG]
```

III. Download Images of Nginx, Node JS, Redis, and MongoDB from Docker Hub

Open a Terminal window and enter the following code. Make sure the Docker engine is running in the background first.

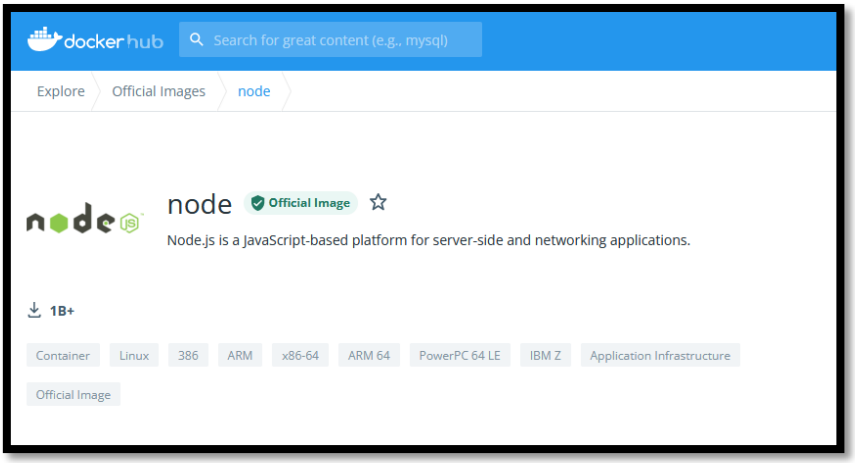
A- Download Nginx - Official Image

```
1  docker pull nginx
```



B- Download Node JS - Official Image

```
2  docker pull node
```

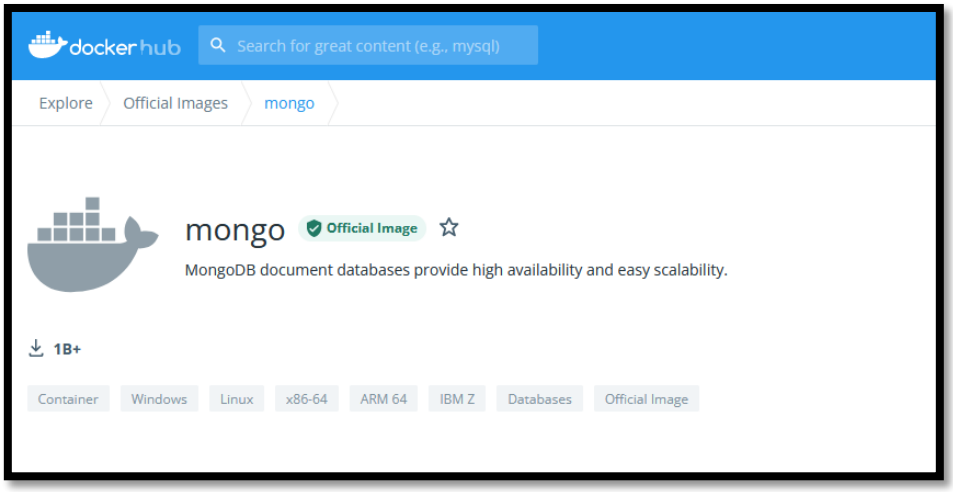


C- Download Redis - Official Image

```
3  docker pull redis
```

D- Download MongoDB - Official Image

```
4  docker pull mongo
```



IV. Install Autocannon Benchmark

For the JavaScript runtime environment Node JS, NPM is the default package manager. It consists of a command-line client, commonly known as NPM, and the NPM registry, an online database of public and paid-for private packages. The client may access the registry, and the NPM website can be used to browse and search for available packages. NPM is in charge of package management and the registry.

First install NPM from [50] and then install autocannon from [47].By inter code in the terminal

```
1  npm i autocannon -g
```

V. Install Postman

The SNAP package is the easiest way to install it on Ubuntu 20.04. Simply run the command below and you're done.

```
1  sudo snap install postman
```

VI. Docker Swarm Mode

Create Docker Swarm Mode by Build a swarm on the host computer and join a node to the cluster.

```
ubu@karar:~/Desktop/look/node-docker-main$ docker node ls
ID                            HOSTNAME   STATUS    AVAILABILITY   MANAGER STATUS   ENGINE VERSION
rm0s1b8o52y0rtgy6okx1b4x2 *   karar      Ready     Active         Leader           20.10.8
mo7ujeg1g2mbo9dfdb6rqaqzs     node1      Ready     Active                          20.10.8
eperoq6qvkucn3du2ufd2b4rd     node2      Ready     Active                          20.10.8
rm1ylpx0kqzayr4wu3lqd59js     node3      Ready     Active                          20.10.8
q3u15mclwa8hd9vjghcic1baf     node4      Ready     Active                          20.10.8
oksvfbk986ruj39x1nga3s0c5     node5      Ready     Active                          20.10.8
```

Terminal of Docker Swarm after Adding All Nodes

There are some steps to begin using the Docker swarm to make a cluster, join a node, and deploy services in this cluster. This can represent these steps:

A- Create Docker Swarm Mode

Build a swarm on the host computers and check to make sure the Docker Engine daemon is running. Open a terminal or SSH session on the system where your management node will be running. To make a new swarm, use the following command:

```
$ docker swarm init --advertise-addr <MANAGER-IP>
```

The “—advertise-addr” option tells the management node to use 192.168.99.100 as its public IP address. The other nodes in the swarm must be able to see the manager's IP address.

B- Join Node to Docker Swarm Mode

The commands to add new nodes to the swarm are included in the output. Depending on the value of the “—token” option, nodes will join as managers or workers.

```
$ docker swarm init --advertise-addr 192.168.99.100
Swarm initialized: current node (dxn1zf6l61qsb1josjja83ngz) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
    192.168.99.100:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
```

You're ready to add worker nodes after you've established a swarm with a management node. Run the command that was created by the "docker swarm init" result from the Create a swarm step to add a worker node to an already existing swarm.

```
$ docker swarm join \
  --token  SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c \
  192.168.99.100:2377

This node joined a swarm as a worker.
```

Launch a terminal and SSH into the host where the management node is operating, then type “docker node ls” to see the workers node.

```
$ docker node ls
ID                            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
03g1y59jwfg7cf99w4lt0f662     worker2   Ready   Active
9j68exjopxe7wfl6yuxml7a7j     worker1   Ready   Active
dxn1zf6l61qsb1josjja83ngz *   manager1  Ready   Active        Leader
```

You're already linked to this node, as shown by the * beside the node ID. The Swarm mode of the Docker Engine assigns a name to each node depending on the machine's hostname. The Management column indicates the swarm's manager nodes. Worker1 and worker2 have an empty status in this column, indicating that they are worker nodes. Only manager nodes may use swarm management commands like “docker node ls”.

C- Deploy Services in Docker Swarm Mode

Deploy service to the swarm. A service can be distributed to a swarm after it has been created. From a terminal SSH into the computer where your management node is running. Execute the command below:

```
$ docker service create --replicas 1 --name helloworld alpine ping docker.com

9uk4639qpg7npwf3fn2aasksr
```

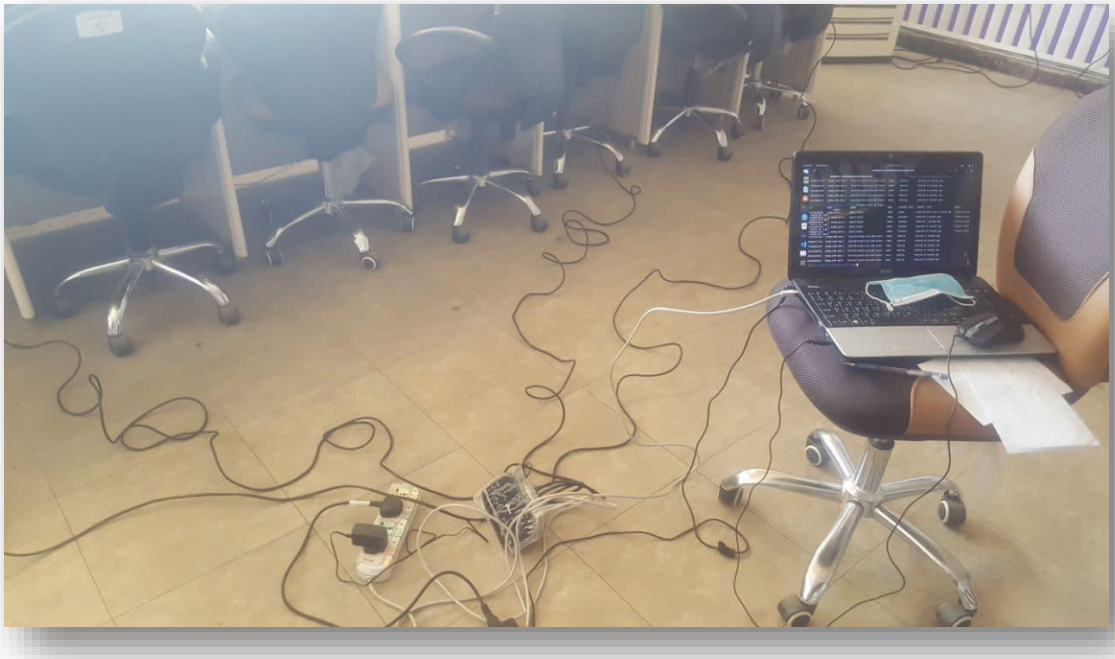
The service is created using the "docker service create" command. The HelloWorld service is identified via the “—name” parameter. The “—replicas” parameter defines a single running instance's intended state. These arguments describe the service as an Alpine Linux container that pings docker.com, and they say it does so by running ping docker. To view a list of currently running services, type “docker service ls”.

```
$ docker service ls

ID            NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

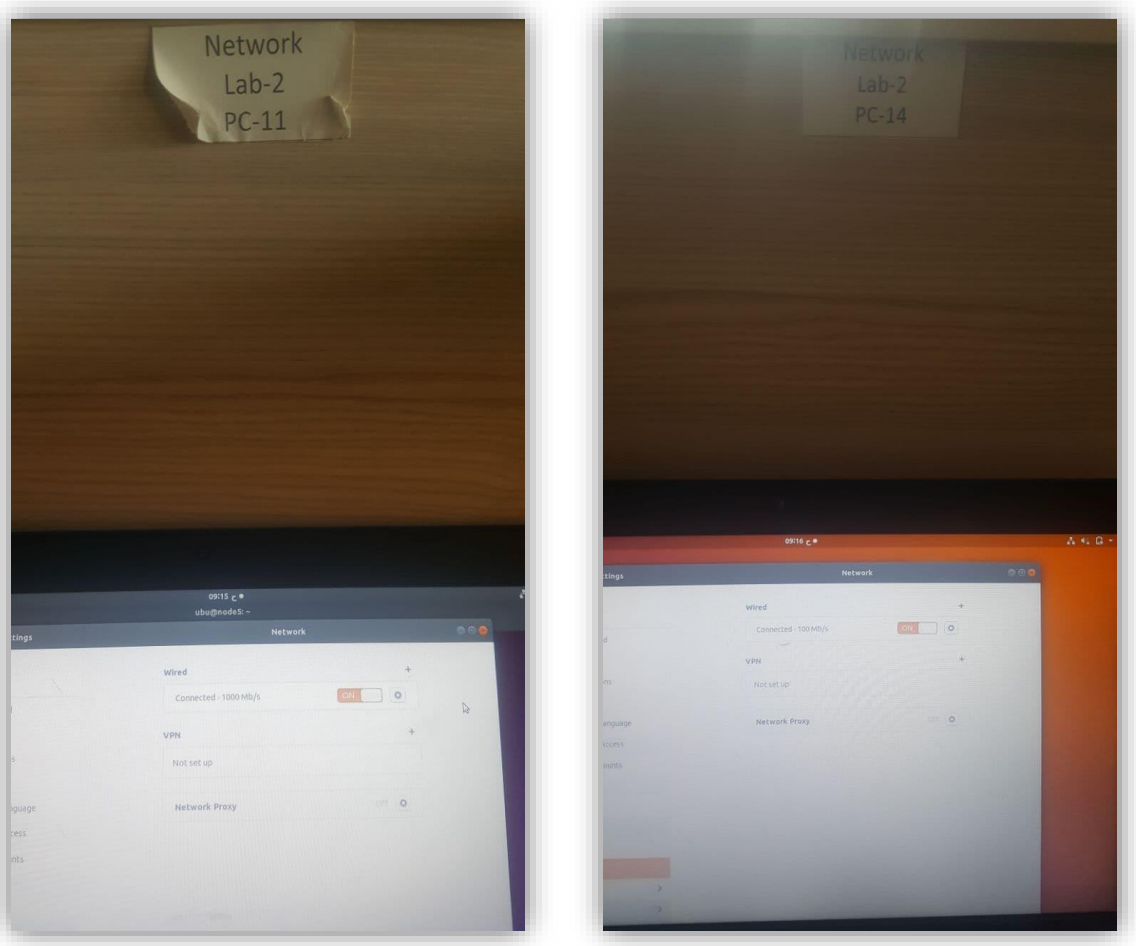
VII. Work Lab

Lab 1 A picture inside a lab with the manager's laptop

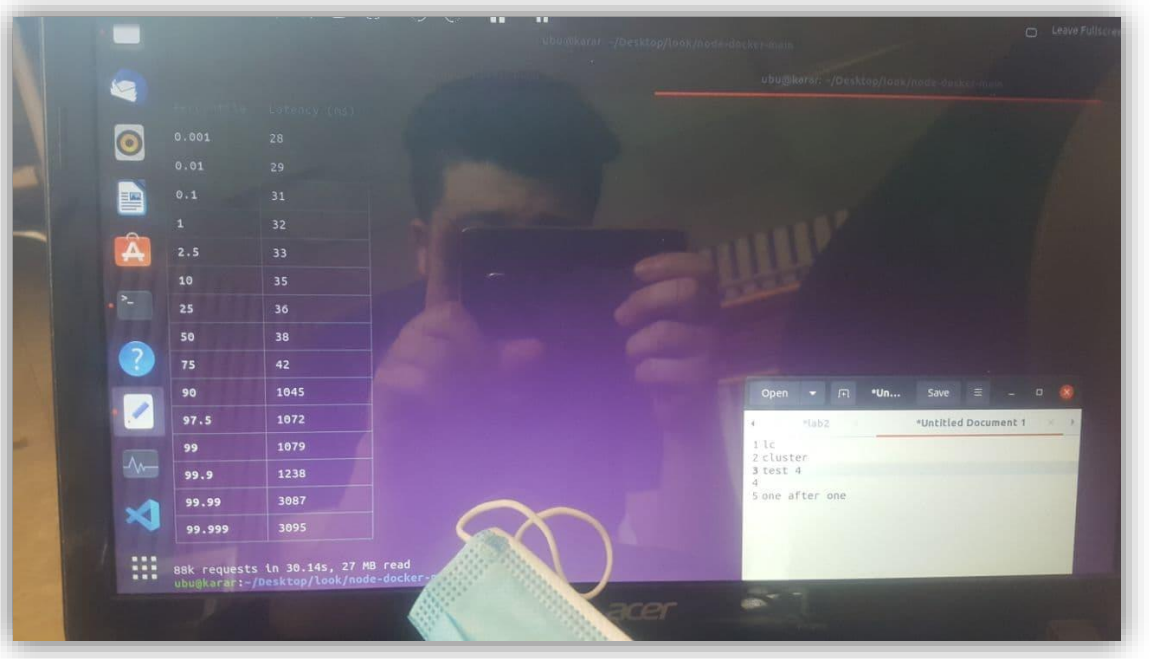


Lab 2 In this picture, the manager's laptop is connected to other worker laptops by a switch inside of a lab.

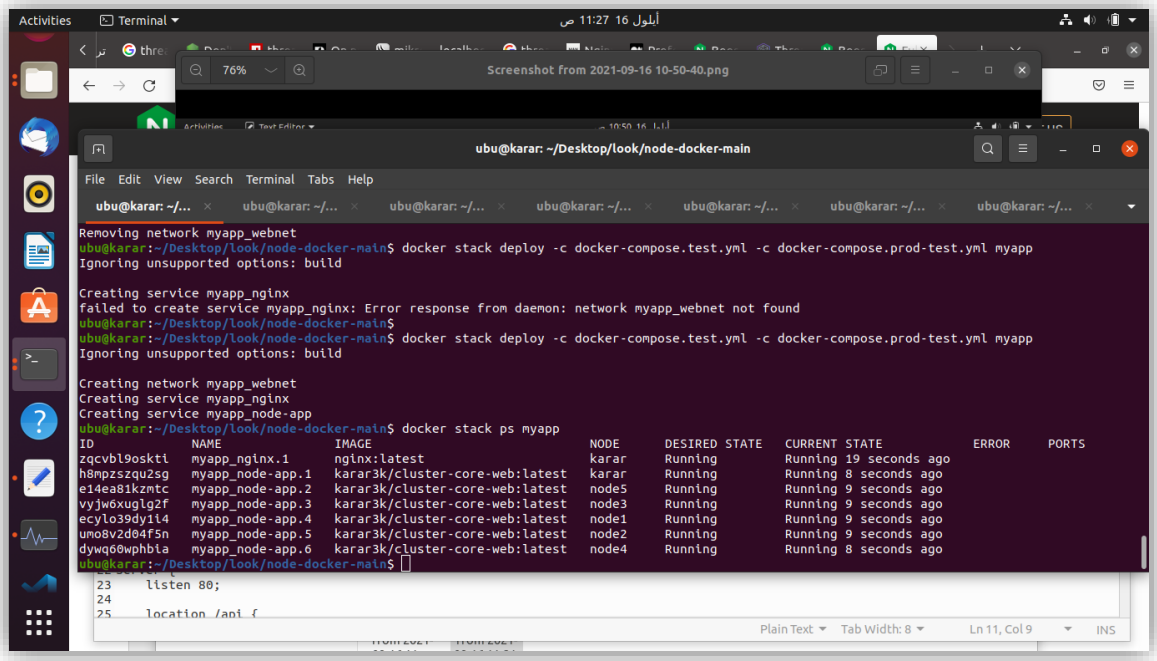
Lab 3 Switch the network so that the manager's laptop can connect to the worker's laptop and the worker's laptop can connect to the manager's laptop by using RJ45 Cable, which is a type of wire.



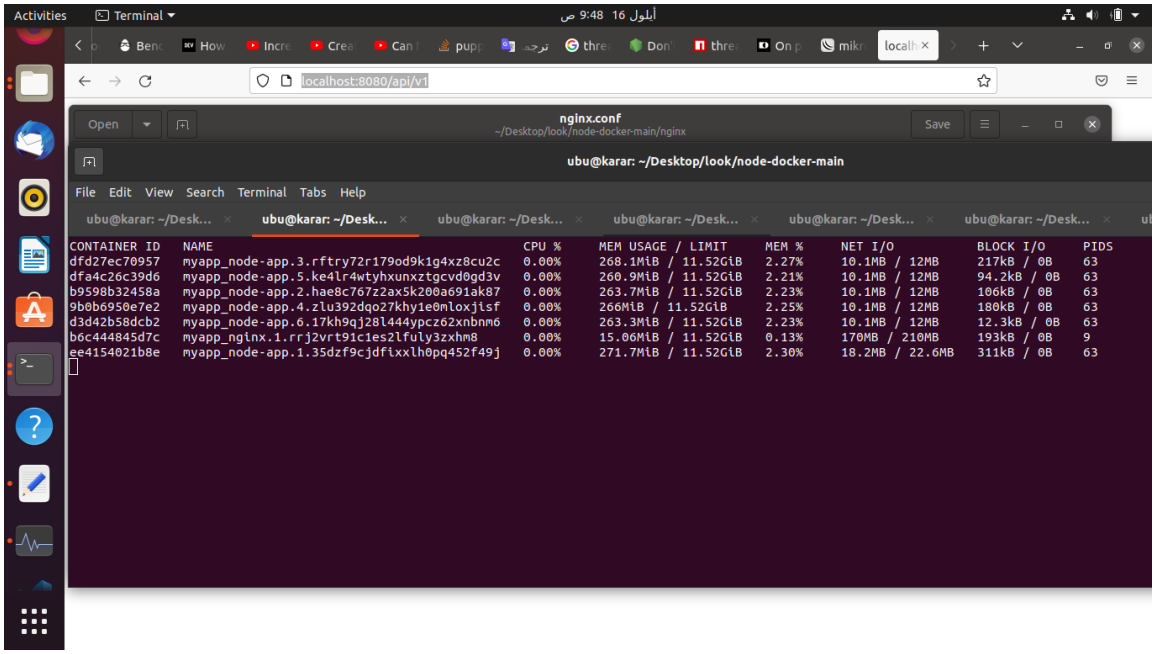
Lab 4 A picture from the lab that shows the working laptops that are in use.



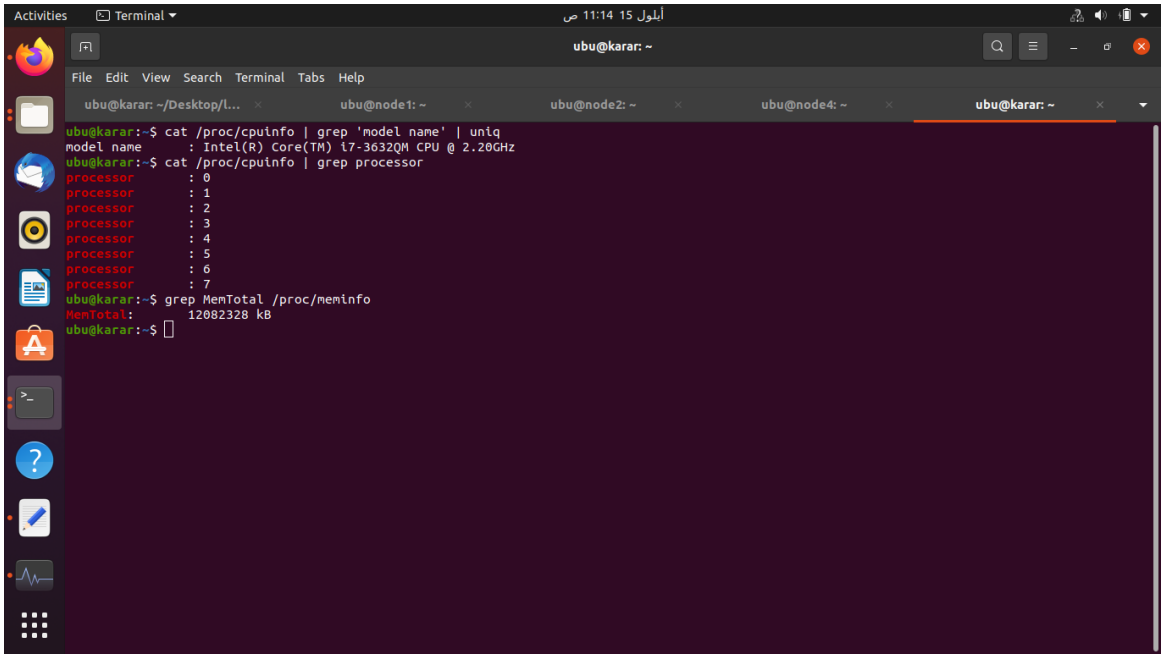
Lab 5 An image shows how the Benchmark Tool autocannon was used to do some of the tests on the proposed system.



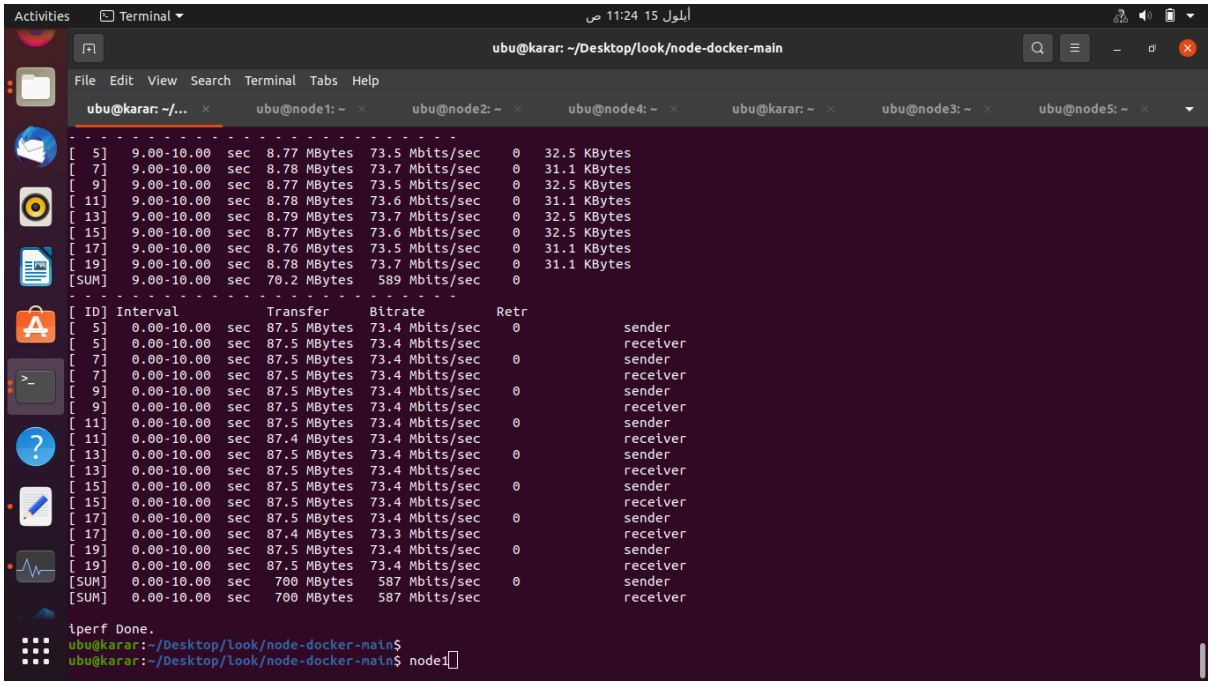
Lab 6 This picture shows how the services are spread out in Docker Swarm.



Lab 7 An image that shows how many services are used in Docker Swarm, as well as how much processor, RAM, and network flow each service uses.



Lab 8 An image of the Manager's laptop, including the number of cores, the core frequency, and the amount of RAM.



Lab 9 An image of a bandwidth network test for node1 is shown.

الخلاصة

يشار إلى عملية تقسيم حركة مرور الشبكة بين عدة خوادم مختلفة باسم موازنة التحميل. هذا يضمن عدم تحميل خادم واحد أكثر من اللازم بالطلبات. كلما زاد توزيع عبء العمل بشكل متساوٍ ، كانت استجابة التطبيق أفضل. بالإضافة إلى ذلك ، فإنه يوسع نطاق اختيار مواقع الويب والتطبيقات التي يمكن للمستهلكين الوصول إليها. تعتبر موازين التحميل ضرورية حتى تعمل التطبيقات الحديثة بشكل صحيح.

اكتشف الرواد الأوائل للحوسبة والإنترنت أن هناك حدودًا مادية لعدد الطلبات التي يمكن معالجتها في فترة زمنية معينة بواسطة الكمبيوتر. لحسن الحظ ، يبدو أن هذه الحدود الفيزيائية تتوسع بمعدل أسي. ومع ذلك ، فإن رغبة الجمهور في الحصول على برامج سريعة ومعقدة تعمل باستمرار على توسيع حدود الآلات ، نظرًا لأننا نجمع مئات الآلاف أو حتى الملايين من الأشخاص عليها.

لذا فإن لدى الشركات خياران متاحان لاستخدامهما في الحلول عند مواجهة الطلب المتزايد من المستخدمين والوصول إلى حد أداء الخادم الذي يستضيف خدمتك: التوسيع (Scale Up) أو التوسع (Scale Out). توجد قيود على الحوسبة المادية عند التوسيع (المعروف أيضًا باسم القياس الرأسي). يتيح لك التوسع، المعروف أيضًا باسم القياس الأفقي ، تفريق حمل الطلب عبر العديد من الخوادم المطلوبة للتعامل مع عبء العمل بطريقة أفقية. يمكن أن يساعد استخدام موازن التحميل في توزيع الطلبات عبر عدد أكبر من الخوادم عند التوسع.

يستخدم الجزء الأول من النظام المقترح خوارزميات موازنة الحمل (Round Robin (RR) و Least Connection (LC)) ونعزز هذه الخوارزميات لأن لها عيوبًا. تضيف خوارزميات التحسين المقترحة وزناً لكل خادم لتحقيق تعزيز لتصبح الخوارزمية الاولى Enhance Weight Round Robin (EWRR) والخوارزمية الثانية Enhance Weight Least Connection (EWLC) مع برمجة المواقع بأستخدم Node JS as

Default (Single-Thread). الجزء الثاني من النظام المقترح هو تحسين الأداء ، واستخدام Multi-Thread في Node JS لمعالجة طلبات كمجموعة ، ومقارنة نتائج الخوارزمية.

نتائج الجزء الأول من استخدام النظام المقترح باستخدام Single-Thread مع (RR) هو 16 ألف طلب و (LC) هو 25 ألف طلب و (EWRR) هو 20 ألف طلب و (EWLC) هو 54 ألف طلب. نتائج الجزء الثاني من النظام المقترح باستخدام Multi-Thread مع (RR) هي 17 ألف طلب و (LC) هي 87 ألف طلب و (EWRR) هي 24 ألف طلب و (EWLC) هي 96 ألف طلب.

جمهورية العراق
وزارة التعليم العالي والبحث العلمي
جامعة بابل كلية تكنولوجيا المعلومات
قسم شبكات المعلومات

تحسين خوارزميات موازنة التحميل لتطوير أداء خادم الويب

الرسالة

مقدمة الى مجلس كلية تكنولوجيا المعلومات جامعة بابل وهي جزء من متطلبات نيل درجة الماجستير في تكنولوجيا المعلومات / شبكات المعلومات

من قبل

كرار حيدر عنون موسى

أشراف

أ.م.د. مهدي صالح نعمة المحنا

2022 م

1443 هـ