Journal of Applied Informatics and Computing (JAIC)
Vol.9, No.2, April 2025, pp. 383~392
e-ISSN: 2548-6861

Optimization of Application Deployment Architecture in Container Orchestration

Mochamad Rizal Fachrudin [1]*, Ahmad Rofiqul Muslikh [2]*
* Department of Information System, Faculty of Information Technology, Universitas Merdeka Malang
mrizalf.email@gmail.com [1], rofickachmad@unmer.ac.id [2]

Article Info

Article history:

Received 2024-11-27
Revised 2024-12-07
Accepted 2025-01-21

Keyword:

Optimization,
Cluster Architecture,
Container Orchestration,
Docker Swarm,
Load Balancing.

ABSTRACT

Container orchestration has become a widely adopted standard for application deployment among medium to large-scale organizations. Docker Swarm is one of the popular container orchestration tools due to its relatively simple configuration. However, if the Docker Swarm cluster architecture is not properly designed, the goal of container orchestration, which is availability, cannot be achieved optimally. Challenges such as centralized traffic on a single node and service dependency on a single node are critical issues that need to be addressed. This study proposes solutions through an experimental approach involving the design, implementation, testing, and evaluation of a Docker Swarm cluster architecture to address these challenges. The results of this study demonstrate that the proposed architecture successfully resolves these issues. Traffic can be distributed more evenly across all nodes. When only one node is available, 5 out of 10 requests can be handled with a response latency of 197.4 ms. With two nodes available, the number of requests handled increases to 7 out of 10, with a response latency of 534.86 ms. The greater the number of available nodes, the more requests can be successfully processed. Services also become more flexible, and capable of running on any node, while offering additional benefits such as dual load balancing through DNS-based load balancing and the default load balancing provided by Docker Swarm's routing mesh. However, limitations such as the need for more complex adjustments and configurations should be considered, especially when implementing this architecture in on-premise environments, to ensure the best adoption and results.

I. Introduction

Containers are a virtualization technique that isolates a system without interfering with others. Unlike hypervisor-based virtualization, applications are more commonly deployed and distributed using containers due to their small size and ease of distribution [1]. There are many containerization tools available, such as Docker, Podman, and others. Docker is one of the most popular platforms and tools currently in use [2]. With Docker, applications can run in various environments because Docker virtualizes the application's runtime into an isolated container [3]. However, when the scale of application deployment grows large, a tool is needed to manage these containers, commonly referred to as container orchestration [4]. Container orchestration has become the standard for medium to large-scale organizations. The primary goals of container orchestration are to automate tasks during application deployment, optimize resource usage, and ensure the high availability of applications [5]. Docker Swarm is one of the widely used container orchestration tools. Docker Swarm excels in deployment speed compared to Kubernetes or Apache Mesos [6]. Docker Swarm offers full support for Docker containerization tools. It operates with two types of nodes: one manager node and multiple worker nodes [7]. Docker Swarm is designed for small-scale deployments, with simpler configurations compared to other orchestration tools like Kubernetes and OpenShift, which require more complex configurations [8]. Unlike Kubernetes, which includes built-in Ingress technology for managing traffic from domains to services, Docker Swarm does not offer a comparable feature. As a result, careful consideration is required when building container orchestration with Docker Swarm. An inadequately designed Docker Swarm cluster architecture can introduce new issues that undermine the goals of orchestration.

In a study conducted by Mohamad Rexa et al., a solution was provided for using load balancing to distribute traffic and address resource issues [9]. When the architecture in that study was implemented, traffic became dependent on the manager node, as it was routed through the manager node first, causing uneven distribution. Research by Stefanus Eko Prasetyo et al. demonstrated the success of load balancing in Docker Swarm, while another study by Dimas Setiawan Afis et al. compared traffic distribution algorithms in Docker Swarm [10], [11]. In both studies, the architecture used also centralizes traffic on a single node, specifically the manager node, by directing traffic to one node running the load balancer before distributing it to the other nodes. This approach is quite risky because the load balancer is statically placed on a single node, meaning that if this node crashes or encounters issues, traffic flow will be disrupted. Such a problem would negatively impact the availability of the cluster. Additionally, since both incoming traffic management and load balancing are handled by the same node, there is a risk of resource exhaustion. Ahmad Rivaldi et al. succeeded in distributing traffic evenly to the worker nodes using an Nginx load balancer [12]. The architecture used in this study still follows the same design, where traffic is directed to the manager node running the load balancer before being distributed to the worker nodes. As a result, the same issue arises: the cluster becomes highly dependent on a single node, and the specifications of that node will significantly impact performance.

Dani Maulana's research utilized three manager nodes to address traffic dependence on a single node [13]. However, in this study, services were run on each manager node using persistent volumes that were not synchronized across nodes. As a result, data was not synchronized between nodes, and when one of the manager nodes failed or crashed, the data stored on that node became inaccessible to other nodes. Another study by Wahyu Aldiwidianto used Keepalived to automatically redirect traffic if a node failed [14]. This method can help solve the problem of node failure, but the challenge is that the application must be deployed globally across each node with local volume storage. As a result, if a server goes down and a request needs data from the failed server, the request cannot be completed.

Based on the issues identified in the architecture of the previous studies, this research aims to optimize the architecture to address the centralization of traffic and node dependency, thereby reducing the potential for overloading a single node and improving the availability of the Docker Swarm cluster. The designed Docker Swarm cluster architecture will adopt the flow of Kubernetes' built-in Ingress technology. This study will also evaluate solutions to the problems identified previously, as well as the strengths and weaknesses of the implemented architecture. This research is expected to provide benefits to both academics and practitioners regarding architectures that can be explored and implemented when building container orchestration clusters, particularly Docker Swarm, or other orchestration technologies with similar issues.

II. METHOD

This study adopts an experimental approach to design, implement, and evaluate the new architecture. The research workflow comprises several stages, as shown in Figure 1.

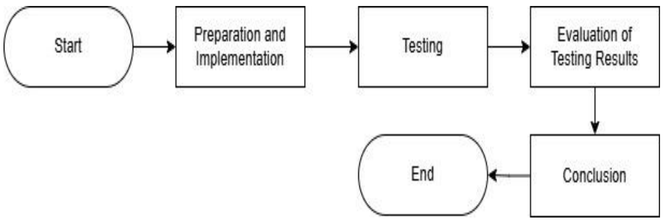


Figure 1. Research Flow

A. Preparation and Implementation

In this stage, all necessary preparations for the research are made. The first requirement is a domain and DNS management system to handle the management and routing of domains and subdomains. Traffic directed to the cluster follows standard routing concepts, where packets are forwarded from the source to the destination through a network [15]. In this study, Cloudflare DNS is used as the DNS management tool. The next requirement is the Docker Swarm cluster, which is set up using three virtual machines (VMs) hosted on a cloud service. The specifications of the VMs are provided in Table 1.

TABLE I
VIRTUAL MACHINE SPECIFICATION

No	Name	Specification
1	VM1	OS Ubuntu, 4GB RAM, 2 vCPU, 80GB Storage
2	VM2	OS Ubuntu, 4GB RAM, 2 vCPU, 80GB Storage
3	VM3	OS Ubuntu, 4GB RAM, 2 vCPU, 80GB Storage

The architectural design for the three VMs serving as cluster nodes is depicted in Figure 2. Based on the designed architecture, the first software requirement for the internal Docker Swarm cluster is a distributed file system. This system is essential for reducing data redundancy and solving data-sharing issues by combining multiple disks into a single volume [16]. Among the various distributed file system options, this study employs GlusterFS, as research by Purwantoro et al. demonstrates that GlusterFS is faster than other systems, such as Ceph [17]. Another key requirement is a reverse proxy tool, which serves as an intermediary between the client represented in this study by Cloudflare and the server, handling all incoming requests on behalf of the server [18].

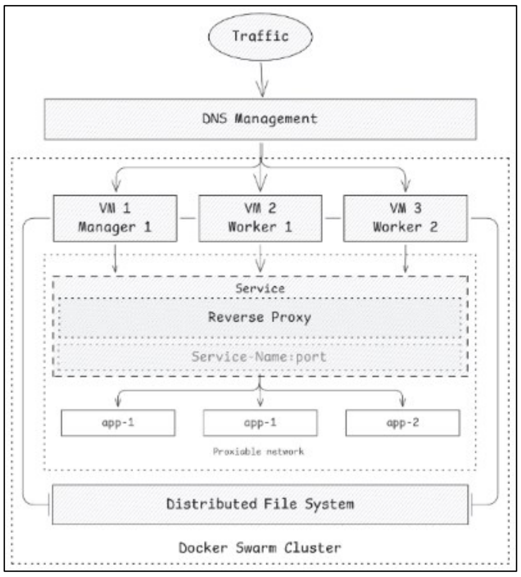


Figure 2. Architecture Design

While there are several reverse proxy tools available, including HAProxy and Traefik, this study chooses Nginx Proxy Manager due to its user-friendly interface and support for GUI-based configuration. The implementation process follows the architecture deployment algorithm outlined below.

TABLE II
IMPLEMENTATION ALGORITHM

```
Algorithm 1. Architecture Implementation
Step 1.
  Initialize the host on a virtual machine
    node_1 = xxx.xxx.xxx.xx
    node_2 = xxx.xxx.xxx.xx
    node_3 = xxx.xxx.xxx.xx
Step 2.
  Initialize the domain being used
    domain = example.com
Step 3.
  Initialize cluster
    create a cluster on node_1
    add node_2, node_3 as a workers
Step 4.
  Initialize distributed file system
    add node_1, node_2, node_3 to the
GlusterFS pool list
    create a shared volume for storage
    add node_2,node_3
    create a folder for the reverse proxy:
      npm_volume = /mnt/npm
Step 5.
  Initialize docker network proxiable
    network = proxiable
    driver = overlay
Step 6.
  Initialize service reverse proxy
    deploy = stack
    service = npm
    network = proxiable
    volumes = npm_volume
Step 7.
  Initialize DNS management
    domain = domain
    IP target = (node_1, node_2, node_3)
Step 8.
  Routing reverse proxy traffic
    domain = domain
    service target = service
```

When applied, this architecture is quite similar to the research conducted by Basel Magableh et al., where a reverse proxy is implemented within the cluster to manage traffic routing [19]. However, this study introduces modifications and additions, with traffic management also being applied outside the cluster, directing it to all nodes. When a client accesses the prepared domain, the traffic is routed through DNS management to the IPs of each node in the Docker Swarm cluster. By directing the domain to all node IPs, DNS management automatically performs load balancing. This is because DNS management is generally equipped with classic load balancing algorithms such as round robin, meaning traffic will be distributed across all IP addresses registered in the DNS management [20]. This method is called DNS-Based Load Balancing, which enables load balancing based on DNS [21]. Since Cloudflare is used for DNS management, the load balancing algorithm follows the available options in Cloudflare. By default, Cloudflare uses the round-robin algorithm for DNS load balancing [22]. Once the traffic is directed to the three node IPs, the reverse proxy receives and forwards the request to the intended service name, as it is within the same network. Docker Swarm, by default, uses the routing mesh strategy, which performs load balancing using the round-robin algorithm [23].

A. Testing

At this stage, testing is conducted using scenarios based on the identified issues. Since the problem with the previous architecture was centralized traffic that had to pass through a single node, the first test focuses on traffic distribution across all nodes. In this phase, several DNS management misconfiguration scenarios will be carried out sequentially until all misconfigurations are resolved, and the minimum estimated traffic resolution during the request test will be calculated. The request test will involve simultaneous operations from different locations. For this initial test, the calculation of the minimum estimated number of resolved requests will be done using Formula 1.

$$P(A) = \frac{n(A)}{n(S)} \qquad (1)$$

The formula represents a probability equation where A is the number of configured nodes, and S is the total number of nodes. This equation is then used to calculate the incoming traffic requests, as shown in Formula 2.

$$E = R \times P(A) \tag{2}$$

From the formula above, E represents the estimated minimum requests that will be completed, while R represents the total number of incoming requests. The second test is the node dependency test, which focuses on the node managing traffic. In this test, a reboot will be performed on the traffic management node, specifically the node running the reverse proxy service, to observe how the cluster, built with the new architecture, handles this issue. This ensures that traffic is not dependent on a specific node. The final test is conducted to observe the traffic distribution from the reverse proxy running the routing mesh with round-robin load balancing. The goal of this test is to evaluate the effectiveness of load balancing and traffic routing when service replicas are scaled up.

B. *Evaluation of Testing Results*

At this stage, the evaluation will be based on the data collected from the testing results. The first evaluation will compare the architectural flow specifications from previous studies to highlight the differences in each study's approach. Next, an evaluation will address the issues that emerged and how they can be solved by the proposed architecture. Lastly, an assessment will be made of the strengths and weaknesses of the newly proposed architecture.

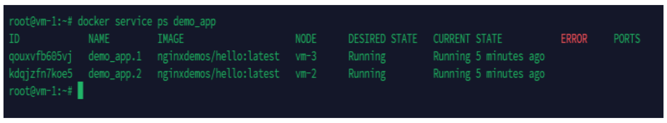
C. *Conclusion*

In the conclusion stage, conclusions will be drawn based on the evaluations conducted. The goal is to provide readers with key insights related to the solutions for the architectural issues discussed earlier. This way, both academics and practitioners can further develop, test, and use it as a reference to determine the suitability of the architecture for the case studies they are currently facing.

III. Results and Discussion

A. *Testing*

The service being run is called 'demo_app' with two replicas, located on the 'proxiable' network, and utilizing a distributed file system. For more details, please refer to Figure 3.



```
root@vm-1:~# docker service ps demo_app
ID             NAME         IMAGE                     NODE      DESIRED STATE   CURRENT STATE           ERROR     PORTS
qouxvfb605vj   demo_app.1   nginxdemos/hello:latest   vm-3      Running         Running 5 minutes ago
kdqjzfn7koe5   demo_app.2   nginxdemos/hello:latest   vm-2      Running         Running 5 minutes ago
root@vm-1:~# 
```

Figure 3. Demo App Service

The first test, related to the issue of traffic distribution, involved the first scenario of misconfiguring the IPs of all nodes, which resulted in an estimated 0% of traffic being resolved. The results are shown in Table 3.

TABLE III
FIRST SCENARIO TESTING

No	Location	Res
1	Roubaix, FR, EU - OVH SAS (AS16276)	failed
2	Milan, IT, EU - Google LLC (AS396982)	failed
3	Toronto, CA, NA - NeuStyle (AS4508)	failed

Based on the results above, the expected estimate has been met, where all requests could not be resolved due to the misconfiguration of all node IPs. In the second scenario, two node IPs were misconfigured. Then, a test was conducted with 10 requests from different regions, and the estimated minimum traffic that could be resolved was 3. The results can be seen in Table 4.

TABLE IV
SECOND SCENARIO TESTING

No	Location	Res
1	Ashburn, US, NA - Oracle Corporation (AS31898)	475 ms
2	Lille, FR, EU - OVH SAS (AS16276)	24 ms
3	Beijing, CN, AS - Shenzhen Tencent Computer Systems Company Limited (AS45090)	1 ms
4	Los Angeles, US, NA - Aptum Technologies (AS13768)	failed
5	Helsinki, FI, EU - Hetzner Online GmbH (AS24940)	344 ms
6	Seoul, KR, AS - Microsoft Corporation (AS8075)	failed
7	Yogyakarta, ID, AS - PT Media Sarana Data (AS55666)	143 ms
8	Dronten, NL, EU - The Infrastructure Group B.V. (AS60404)	failed
9	Sydney, AU, OC - Oracle Corporation (AS31898)	failed
10	Brussels, BE, EU - M247 Europe SRL (AS9009)	failed
Average Latency		197,4 ms

The results above show that 5 requests were completed, while 5 other requests failed, with an average response latency of 197.4 ms. The successful requests cover various geographic regions around the world, indicating network performance variations based on the server's geographical location. The estimate obtained was higher than the minimum resolution estimate, which was only 3 requests. In the third scenario, there was only one node misconfiguration. The minimum estimated number of requests that could be resolved was 6. Then, 10 requests from different regions were tested, and the results are shown in Table 5.

TABLE V
THIRD SCENARIO TESTING

No	Location	Res
1	London, GB, EU - OVH SAS (AS16276)	238 ms
2	Jakarta, ID, AS - PT JEMBATAN CITRA NUSANTARA (AS23951)	150 ms
3	Chengdu, CN, AS - Shenzhen Tencent Computer Systems Company Limited (AS45090)	1387 ms
4	Falkenstein, DE, EU - Hetzner Online GmbH (AS24940)	634 ms
5	Helsinki, FI, EU - Hetzner Online GmbH (AS24940)	358 ms
6	Berlin, DE, EU - Google LLC (AS396982)	failed
7	Belgrade, RS, EU - mCloud doo (AS35779)	failed
8	Tokyo, JP, AS - xTom Japan Co., Ltd. (AS3258)	127 ms
9	Bucharest, RO, EU - M247 Europe SRL (AS9009)	850 ms
10	Paris, FR, EU - Hivane Association (AS34019)	failed
Average Latency		534,86 ms

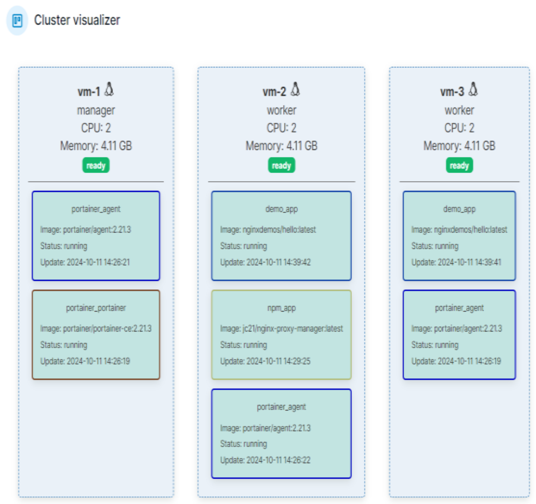
Based on the results above, 7 out of 10 requests were completed, with an average response latency of 534.86 ms. This is slightly higher than the minimum estimated completion of requests. In the last scenario, Scenario 4, there was no misconfiguration of node IPs. The expected estimate for requests that could be completed was all incoming traffic. After conducting a test with 10 requests from different regions, the results were as shown in Table 6.

TABLE VI
FOURTH SCENARIO TESTING

No	Location	Res
1	Halifax, CA, NA - Free Range Cloud Hosting Inc. (AS53356)	661 ms
2	Mahoba, IN, AS - ReadyDedis LLC (AS140543)	923 ms
3	Matsuyama, JP, AS - ARTERIA Networks Corporation (AS17506)	154 ms
4	Vladivostok, RU, AS - PortTelekom LLC (AS34470)	2908 ms
5	Dallas, US, NA - Catalyst Host LLC (AS393336)	487 ms
6	Mumbai, IN, AS - Google LLC (AS396982)	243 ms
7	Warsaw, PL, EU - Liberty Global B.V. (AS6830)	858 ms
8	Ho Chi Minh City, VN, AS - Zenlayer Inc (AS21859)	193 ms
9	Amsterdam, NL, EU - Psychz Networks (AS40676)	229 ms
10	Moscow, RU, AS - Datacheap LLC (AS16262)	400 ms
Average Latency		705,6 ms

Based on the results above, in the final scenario, all requests were completed as expected, with a response latency of 705.6 ms. This approach improves the availability and stability of the cluster. In the case of a single node failure, 7 out of 10 requests were still completed, and in the case of two node failures, 5 out of 10 requests were successfully processed, preventing a total traffic halt in the cluster. Each node can handle incoming traffic, resulting in a more balanced load distribution and reducing the potential risk of resource exhaustion. Additionally, it provides time for the technical team to resolve issues without total downtime. Therefore, this architecture can be a solution for companies with limited resources that still prioritize uptime, such as e-commerce, media streaming, or SaaS (Software as a Service) providers.

In this second test, a reboot will be conducted as a simulation of a node crash, specifically on the node running the reverse proxy, to see if the reverse proxy can be recreated on another node without causing the configuration to fail and crash other services. An initial overview can be seen in Figure 4.



vm-1
manager
CPU: 2
Memory: 4.11 GB

portainer_agent
Image: portainer/agent:2.21.3
Status: running
Update: 2024-10-11 14:26:21

portainer_portainer
Image: portainer/portainer-ce:2.21.3
Status: running
Update: 2024-10-11 14:26:19

vm-2
worker
CPU: 2
Memory: 4.11 GB

demo_app
Image: nginxdemos/hello:latest
Status: running
Update: 2024-10-11 14:39:42

npm_app
Image: jc21/nginx-proxy-manager:latest
Status: running
Update: 2024-10-11 14:29:25

portainer_agent
Image: portainer/agent:2.21.3
Status: running
Update: 2024-10-11 14:26:22

vm-3
worker
CPU: 2
Memory: 4.11 GB

demo_app
Image: nginxdemos/hello:latest
Status: running
Update: 2024-10-11 14:39:41

portainer_agent
Image: portainer/agent:2.21.3
Status: running
Update: 2024-10-11 14:26:19

Figure 4. Cluster Visualization Before Crash

From the cluster visualization above, node 2, which is running the reverse proxy, was rebooted, and the resulting cluster visualization is shown in Figure 5.

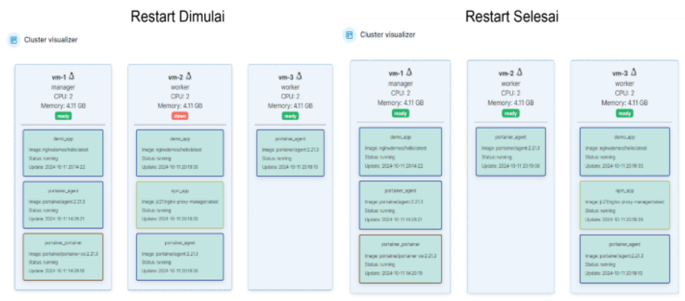
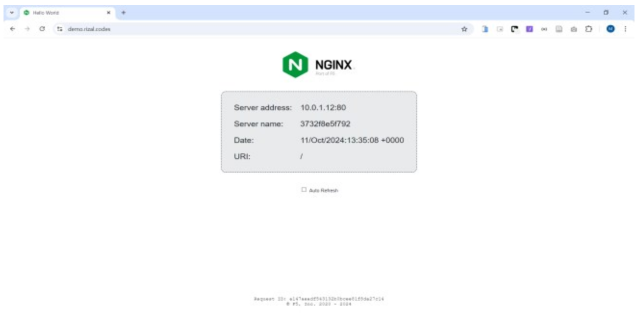


Figure 5. Cluster Visualization After Crash

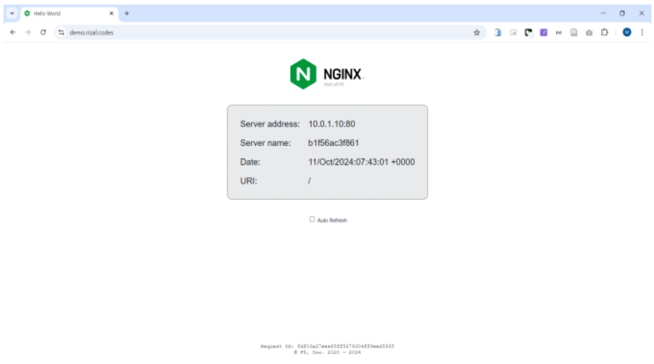
From the image above, it can be seen that when node 2 goes down, a reverse proxy service is created on another node. When a request is made to the domain https://demo.rizal.codes, the service continues to operate normally without any issues. For further clarity, please refer to Figure 6.


Server address: 10.0.1.12:80
Server name: 3732f8e5f792
Date: 11/Oct/2024:13:35:08 +0000
URI: /

Request ID: [illegible]
© F5, Inc. 2020 - 2024

Figure 6. Request Response

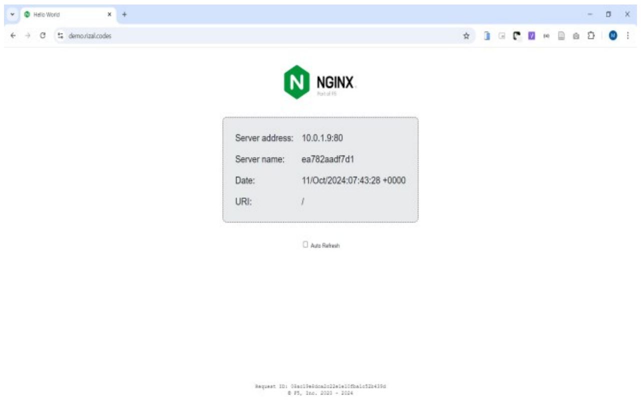
Based on the results above, the domain continues to function normally when requests are made, without requiring any reconfiguration. By implementing the reverse proxy as a service within the cluster, with its configuration distributed across nodes, significant benefits are gained in maintaining the cluster's continuity. Traffic from DNS management is routed to the reverse proxy service, which is automatically migrated to another node if issues arise. This ensures continuous application accessibility, improves reliability, and provides resilience against system failures, ensuring high availability for critical applications. The final test was conducted to assess the success of the routing mesh with the load balancer. With the service running two replicas on different nodes, the first request was made, and the server address and server name were obtained, as shown in Figure 7.


Server address: 10.0.1.10:80
Server name: b1f56ac3f861
Date: 11/Oct/2024:07:43:01 +0000
URI: /

Request ID: [illegible]
© F5, Inc. 2020 - 2024

Figure 7. First Request Response

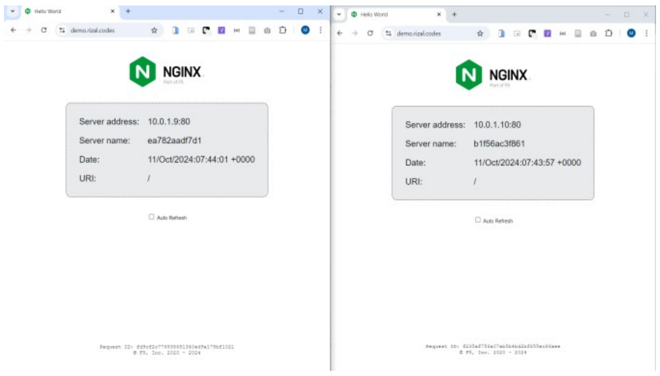
The server address obtained is 10.0.1.10:80 with the server name b1f56ac3f861. Then, on the next request, using the same browser, the result is shown in Figure 8.


Server address: 10.0.1.9:80
Server name: ea782aadf7d1
Date: 11/Oct/2024:07:43:28 +0000
URI: /

Request ID: [illegible]
© F5, Inc. 2020 - 2024

Figure 8. Second Request Response

From the results above, different server addresses and server names were obtained. Then, for the third and fourth request trials, using the same browser and connection, the results are shown in Figure 9.


Server address: 10.0.1.9:80
Server name: ea782aadf7d1
Date: 11/Oct/2024:07:44:01 +0000
URI: /

Request ID: [illegible]
© F5, Inc. 2020 - 2024

Server address: 10.0.1.10:80
Server name: b1f56ac3f861
Date: 11/Oct/2024:07:43:57 +0000
URI: /

Request ID: [illegible]
© F5, Inc. 2020 - 2024

Figure 9. Third Request Response

Traffic management with Nginx Proxy Manager successfully utilizes the default routing of Docker Swarm's routing mesh, effectively providing different server information for each request based on the running replicas. The implementation of load balancing with the round-robin algorithm ensures even traffic distribution, improving overall system performance. By using service names for routing, the need for manual IP node configuration is eliminated, simplifying infrastructure management. This approach allows service replication to meet high availability needs without overloading any node, while also supporting better scalability and reducing the risk of configuration errors. The combination of these features makes the solution reliable and efficient for modern container-based architectures.

A. Evaluation of Testing Results

Before evaluating further the tests of the solution that has been implemented, it is important to understand the specifications of the architectural flow in previous studies. A comparison of the architectural flows can be seen in Table 7.

TABLE VII
COMPARISON OF ARCHITECTURE FLOW SPECIFICATIONS

Researcher	Architecture Flow Specifications
[9]	Traffic from the internet arrives and is directed to Nginx on the manager node. Once the manager node receives the traffic, it performs load balancing using Nginx, based on the memory usage of each connected worker node. After determining the worker node, the traffic is sent to the destination container, and the result is returned to the client.
[11]	Traffic from the internet arrives and is directly directed to the node running the Nginx Load Balancer. Once the traffic is received, load balancing is performed, directing the traffic to the web server containers.
[10]	Traffic from the internet arrives and is directed to the manager node running the Nginx service. Once the node receives the traffic, load balancing is performed on the backend service using Nginx with the least connection and round-robin algorithms. The worker node handling the service then receives the traffic and returns the result to the client.
[12]	Traffic from the internet arrives and is directed to Nginx on the manager node. Once the manager node receives the traffic, load balancing is performed by Nginx, directing the traffic to five target services. The worker node receiving the traffic then returns the result of the service.
[13]	Traffic from the internet does not know which manager node it will be directed to. Once the traffic enters the cluster, the node receiving the traffic will return the requested result to the client.
[14]	Traffic from the internet is directed to the virtual IP managed by Keepalived. Keepalived performs health checks on all nodes and directs the traffic to the node that is functioning normally. The primary priority is given to the node configured as the master. If the master node fails, the traffic will be forwarded to the backup node. The service on the node will then return the result to the client.
Author	Traffic from the internet is directed to DNS management. From DNS management, load balancing is performed using the round-robin algorithm, directing traffic to all IP addresses of the registered nodes. Once a node receives the traffic, it will be directed to the reverse proxy service, regardless of which node it resides on. After the reverse proxy receives the traffic, a routing mesh is performed on the target service, utilizing the round-robin load balancing algorithm. The node running the service will then return the requested result to the client.

After thoroughly understanding the differences in the workflow of each architecture, an evaluation of the results from testing the new architecture was carried out, using the problem scenarios that were encountered in the previous architecture as a reference. This evaluation aimed to assess the effectiveness of the new architecture in addressing the issues that arose in the previous setup. The detailed results of this evaluation, which include a comparison of the performance and problem-solving capabilities of the new architecture, are presented in Table 8. This table provides a comprehensive breakdown of how the new solution tackles the challenges faced in the previous architecture, offering insights into the improvements and adjustments made.

TABLE VIII
EVALUATION OF ARCHITECTURE SOLUTIONS

No	Problem	Solution
1	Traffic Distribution	For traffic distribution, all node IPs are added to the DNS management system to enable DNS-based load balancing, which in this case uses the round-robin algorithm from Cloudflare. This solution evenly distributes traffic across all nodes. In the 3-node scenario, if 1 node fails, approximately ⅔ of the traffic can still be processed, and testing shows that 7 out of 10 requests were successfully handled. If 2 nodes fail, the estimate is that ⅓ of the traffic will be processed, with testing recording 5 out of 10 requests completed. This architecture is effective in distributing traffic and preventing resource exhaustion on the nodes.
2	Node Dependability	Previous research has shown that architectures relying on a single node to receive traffic (e.g., load balancers) increase the risk if that node crashes or reboots, as the service configuration is centralized only on that node. In the new architectural solution, the traffic receiver is a reverse proxy running as a service with configuration storage distributed across nodes. If the node running the reverse proxy encounters an issue, the service is automatically transferred to another node with the same configuration, ensuring the application remains accessible. This solution improves high availability, reduces cluster

		downtime, and eliminates dependence on a single node.
3	Routing and Traffic Load Balancing	Since the reverse proxy operates on the same network as the application service, routing can be done using the service name without the need to manually define the node's IP address, meaning the reverse proxy does not need to know the node's location. Additionally, services do not have to be replicated on every node, allowing replication to be adjusted according to high availability requirements and organizational standards. This architectural solution also enables the reverse proxy to perform automatic load balancing using a routing mesh with a round-robin algorithm, as demonstrated in previous research and testing.

Although this architecture successfully addresses the issues of the previous one, it comes with its own set of strengths and weaknesses. The evaluation of these aspects is provided in Table 9. This table outlines the advantages and potential limitations of the newly implemented architecture.

TABLE IX
STRENGTHS AND WEAKNESSES OF THE ARCHITECTURE

No	Strength	Weakness
1	Incoming traffic is more evenly distributed across all nodes, reducing the risk of resource exhaustion.	A DNS management tool that supports DNS-based load balancing is needed to distribute traffic evenly across each node.
2	Reduces dependency or concentration of traffic on a specific node running the reverse proxy/load balancer.	The DNS-based load balancing algorithm depends on the DNS management tool being used.
3	Application service replicas do not need to be evenly distributed across all nodes and can be adjusted according to needs.	Configuration is slightly more complex due to the need for additional setup.
4	The reverse proxy configuration does not need to know which node the application service is running on.	When implemented in an on-premise environment, the configuration becomes more complex and requires further adjustments.
5	Two layers of load balancing are achieved: DNS-based load balancer and routing mesh load balancer.	

The table above outlines the advantages and disadvantages of the newly designed architecture. From the perspective of its strengths, this architecture uses DNS management for routing across all nodes, ensuring that client traffic is distributed evenly among all registered nodes. The traffic distribution leverages a round-robin algorithm, meaning that during high traffic volumes, all traffic will be balanced evenly across all nodes. This equal distribution reduces the potential for overloading a single node, as no single node is solely responsible for handling the incoming traffic. Testing results also show that traffic handling improves as the number of available nodes increases. In other words, higher node scalability or the implementation of multi-master clustering significantly increases the chances of successfully managing incoming traffic.

Additionally, the traffic management within the cluster is handled by Nginx Proxy Manager, which operates as a service, enabling it to run flexibly on any node. Furthermore, the data stored in the distributed volume ensures that the Nginx Proxy Manager can operate on another node without losing its configuration. This approach eliminates the need to deploy the service on every node, as was required in some previous studies.

The Docker network proxiable configuration also simplifies routing additions from domains to target services. The key advantage here is that the target does not need to be defined using the IP address of the node hosting the application service; instead, only the service name is required. Nginx Proxy Manager does not need to know which node is running the application service—it simply calls the service name. When the application service is replicated, Nginx Proxy Manager automatically uses the default load balancing from Docker Swarm's routing mesh, applying a round-robin algorithm to distribute traffic among the service replicas. This architecture enables the cluster to benefit from dual load balancing using the round-robin algorithm: the first from DNS-based load balancing, which distributes traffic from the internet/clients to all registered nodes, and the second from Docker Swarm's routing mesh, which distributes traffic from Nginx Proxy Manager to the service replicas. This dual load balancing process enhances load distribution by balancing the load at both the node and service levels within the cluster.

However, because this study applies the cluster environment in a cloud service setting, challenges may arise when implementing it in on-premise environments, as highlighted in the disadvantages listed in Table 8. One of the main issues for some organizations could be selecting a DNS management solution that supports DNS-based load balancing. However, as demonstrated in this study, Cloudflare DNS management, which supports DNS-based load balancing, offers this feature for free. This makes it a cost-effective choice for DNS management, although the load balancing algorithm is limited to round-robin.

Another challenge lies in the cluster configuration, which is slightly more complex than previous architectures. Unlike earlier setups, this architecture requires configuring GlusterFS volumes, ensuring DNS management supports DNS-based load balancing, and ensuring all running services use the same Docker network. While these configurations are not overly complicated, the previous architecture is still simpler. Moreover, when building the cluster in an on-premise environment, the complexity increases. In a cloud setting, each virtual machine automatically receives a public IP address. However, in an on-premise setup, acquiring public IPs for all nodes is necessary for DNS management to route traffic to each node. This introduces additional costs since renting public IPs can be expensive. If only one public IP is used, additional equipment such as routers is required to route traffic through a single IP. These challenges highlight the trade-offs and should be carefully considered when implementing the architecture, particularly in on-premise environments.

IV. CONCLUSIONS

Based on the results of this study, it can be concluded that this architecture successfully addresses the issues that arose in the architecture of previous studies, namely the concentration of traffic on a single node and the dependence on one node running the reverse proxy service. Traffic was successfully handled for 5 out of 10 requests with a response latency of 197.4 ms across different locations, even when only one node was available. This improved to 7 out of 10 requests with a response latency of 534.86 ms when two nodes were available, with the success rate increasing as more nodes became available. Additionally, this architecture enables the reverse proxy service, used as a router and load balancer, to operate flexibly on any node, with data distributed across nodes. Another benefit is the dual load balancing achieved through the round-robin algorithm of DNS-based load balancing and the load balancing provided by the routing mesh, ensuring traffic is evenly distributed both to cluster nodes and services within the cluster. However, these weaknesses must still be considered to maximize the adoption results.

In future similar research, the author suggests further exploring the architecture used in this study, such as load balancing methods, tools used, or even the implementation of multi-manager clusters to enhance high availability. Additionally, the author also recommends conducting more in-depth testing aligned with the objectives of container orchestration itself, to provide a more complex comparison and evaluation, which can be used as a reference for readers in more complex real-world case studies.

REFERENCES

[1] RedHat, "Containers vs VMs." Accessed: Nov. 14, 2024. [Online]. Available: https://www.redhat.com/en/topics/containers/containers-vs-vms

[2] Stack Overflow, "Technology | 2024 Stack Overflow Developer Survey." Accessed: Oct. 07, 2024. [Online]. Available: https://survey.stackoverflow.co/2024/technology/

[3] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance Evaluation of Docker Container and Virtual Machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020, doi: 10.1016/j.procs.2020.04.152.

[4] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal Pod Autoscaling in Kubernetes for Elastic Container Orchestration," *Sensors*, vol. 20, no. 16, p. 4621, 2020, doi: 10.3390/s20164621.

[5] IBM, "Apa yang dimaksud dengan orkestrasi kontainer." Accessed: Nov. 14, 2024. [Online]. Available: https://www.ibm.com/id-id/topics/container-orchestration

[6] A. Pankowski and P. Powroźnik, "Comparison of application container orchestration platforms," *Journal of Computer Sciences Institute*, vol. 29, pp. 383–390, Dec. 2023, doi: 10.35784/jcsi.3823.

[7] A. Farshteindiker and R. Puzis, "Leadership Hijacking in Docker Swarm and Its Consequences," *Entropy*, vol. 23, no. 7, p. 919, 2021, doi: 10.3390/e23070914.

[8] L. Mercl and J. Pavlik, "The Comparison of Container Orchestrators," in *Third International Congress on Information and Communication Technology*, X.-S. Yang, S. Sherratt, N. Dey, and A. Joshi, Eds., Singapore: Springer Singapore, 2019, pp. 677–685. doi: 10.1007/978-981-13-1165-9_62.

[9] M. R. Mei Bella, M. Data, and W. Yahya, "Implementasi Load Balancing Server Web Berbasis Docker Swarm Berdasarkan Penggunaan Sumber Daya Memory Host," *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, vol. 3, no. 4, pp. 3478–3487, Jan. 2019.

[10] D. S. Afis, M. Data, and W. Yahya, "Load Balancing Server Web Berdasarkan Jumlah Koneksi Klien Pada Docker Swarm," *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, vol. 3, no. 1, pp. 925–930, Jan. 2019.

[11] S. E. Prasetyo and A. Wijaya, "Analisis Load Balancing Menggunakan Docker Swarm," *CoMBInES*, vol. 1, no. 1, pp. 527–538, 2021.

[12] A. Rivaldi, U. Darusalam, and D. Hidayatullah, "Perancangan Multi Node Web Server Menggunakan Docker Swarm dengan Metode Highavability," *Jurnal Media Informatika Budidarma*, vol. 4, p. 529, Jul. 2020, doi: 10.30865/mib.v4i3.2147.

[13] D. M. Ferdiansyah and A. Prihanto, "Analisis Perbandingan Kinerja High Availability Pada Cluster Docker Swarm Dan K3S," *Journal of Informatics and Computer Science*, vol. 06, no. 2, pp. 210–218, 2024.

[14] W. Aldiwidianto, G. Lanang, and E. Prismana, "Analisis Perbandingan High Availibility Pada Web Server Menggunakan Orchestration Tool Kubernetes Dan Docker Swarm," *Journal of Informatics and Computer Science*, vol. 05, no. 2, pp. 138–148, 2023, doi: 10.26740/jinacs.v5n02.p138-148.

[15] R. D. Marcus, A. S. Ilmananda, L. Indana, and H. A. Aswari, "Optimalisasi Manajemen Jaringan pada Laboratorium Komputer Melalui Implementasi Remote Installation Services," *Jurnal MediaTIK*, vol. 6, no. 3, pp. 79–85, 2023, doi: 10.26858/jmtik.v6i3.51964.

[16] J.-Y. Lee, M.-H. Kim, S. A. Raza Shah, S.-U. Ahn, H. Yoon, and S.-Y. Noh, "Performance Evaluations of Distributed File Systems for Scientific Big Data in FUSE Environment," *Electronics*, vol. 10, no. 12, p. 1471, 2021, doi: 10.3390/electronics10121471.

[17] S. P. E.S.G.S, "Perbandingan Kinerja Clustered File System pada Cloud Storage menggunakan GlusterFS dan Ceph," *INOVTEK Polbeng - Seri Informatika*, vol. 7, p. 319, Nov. 2022, doi:

10.35314/isi.v7i2.2753.

[18] P. Satya Saputra, P. Aditya Pratama, and L. Putu Ary Sri Tjahyanti, “Perancangan Dan Komparasi Web Server Nginx Dengan Web Server Apache Serta Pemanfaatan Reverse Proxy Server Pada Nginx,” *Jurnal Komputer dan Teknologi Sains (KOMTEKS)*, vol. 2, no. 1, pp. 16–21, 2023.

[19] B. Magableh and M. Almiani, “A Self Healing Microservices Architecture: A Case Study in Docker Swarm Cluster,” in *Advanced Information Networking and Applications*, L. Barolli, M. Takizawa, F. Xhafa, and T. Enokido, Eds., Cham: Springer International Publishing, 2020, pp. 846–858. doi: 10.1007/978-3-030-15032-7_71.

[20] J. Ruohonen, “Measuring Basic Load-Balancing and Fail-Over Setups for Email Delivery via DNS MX Records,” in *2020 IFIP Networking Conference (Networking)*, Institute of Electrical and Electronics Engineers, 2020, pp. 815–820.

[21] K. Schomp, O. Bhardwaj, E. Kurdoglu, M. Muhaimen, and R. K. Sitaraman, “Akamai DNS: Providing Authoritative Answers to the World’s Queries,” in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, in SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 465–478. doi: 10.1145/3387514.3405881.

[22] Cloudflare, “Round-robin DNS.” Accessed: Nov. 14, 2024. [Online]. Available: https://developers.cloudflare.com/dns/manage-dns-records/how-to/round-robin-dns/

[23] M. Ileana, O. Maria Ioana, and C. Marian, “Using Docker Swarm to Improve Performance in Distributed Web Systems,” in *17th International Conference on Development And Application Systems*, Institute of Electrical and Electronics Engineers, 2024, pp. 1–6. doi: 10.1109/DAS61944.2024.10541234.