Intelligent Automation & Soft Computing
DOI:10.32604/iasc.2022.024317
Article

Tech Science Press

Ant Colony Optimization-based Light Weight Container (ACO-LWC) Algorithm for Efficient Load Balancing

K. Aruna[1,*] and G. Pradeep[2]

[1]Department of Information Technology, A.V.C. College of Engineering, Mannampandal, Mayiladuthurai, 609305, India
[2]Department of Computer Applications, A.V.C. College of Engineering, Mannampandal, Mayiladuthurai, 609305, India
*Corresponding Author: K. Aruna. Email: avcce.arunait@avccengg.net
Received: 13 October 2021; Accepted: 27 December 2021

Abstract: Container technology is the latest lightweight virtualization technology which is an alternate solution for virtual machines. Docker is the most popular container technology for creating and managing Linux containers. Containers appear to be the most suitable medium for use in dynamic development, packaging, shipping and many other information technology environments. The portability of the software through the movement of containers is appreciated by businesses and IT professionals. In the docker container, one or more processes may run simultaneously. The main objective of this work is to propose a new algorithm called Ant Colony Optimization-based Light Weight Container (ACO-LWC) load balancing scheduling algorithm for scheduling various process requests. This algorithm is designed such that it shows best performance in terms of load balancing. The proposed algorithm is validated by comparison with two existing load balancing scheduling algorithms namely, least connection algorithm and round robin algorithm. The proposed algorithm is validated using metrics like response time (ms), mean square error (MSE), node load, largest Transactions Per Second (TPS) of cluster (fetches/sec), average response time for each request (ms) and run time (s). Quantitative analysis show that the proposed ACO-LWC scheme achieves best performance in terms of all the metrics compared to the existing algorithms. In particular, the response time for least connection, round robin and the proposed ACO-LWC algorithm are 58, 60 and 48 ms respectively when 95% requests are finished. Similarly, the error for scheduling 120 requests using least connection, round robin and the proposed ACO-LWC algorithm are 0.15, 0.11 and 0.06 respectively.

Keywords: Docker; containerization; ant colony optimization; light weight container; load balancing

1 Introduction

In the last few years, server virtualization is popularly being used in the field of information technology. Virtualization acts as a valuable tool for running multiple operating systems in cloud. It facilitates the implementation of various virtual machines on a single physical hardware. The common drawbacks of

virtualization include, increased size, unstable performance, longer time to boot up, etc [1]. To avoid these drawbacks containers are popularly being used. These containers avoid the time required for the configuration of libraries. In these containers the main components are the images. The image file contains the code required and the necessary libraries required for the operation of an application [2].

The docker images are collectively present in Docker Hub. These repositories contain the docker images in an organized manner. The docker repositories can be of two types. The first is the private docker repository and the second is the public docker repository. They are also split into official repositories and community repositories [3]. The official repositories are more popularly used as they contain the public certified images. These images are used as a base platform for other images. Hence, maintaining the security of the official repository images is mandatory. On the other hand, the community repositories can be generated by any individual [4]. Containers do not have any guest operating system. This helps to reduce the overall overload of the system. In real-time, the container system is combined with the Ethernet systems to achieve scalable performance. This helps to create automation along with the internet facilities. These automation processes can be easily controlled using machine learning algorithms [5]. Recently, docker containers are known for providing services to multiple applications using shared hosts. These docker containers are much lighter compared to the virtual machine devices. The usage of docker containers aid the developers to share many applications [6]. Containers are maintained using container management systems. These services render Application Programming Interface (API) for the management of the life cycle. This comprises of multiple levels. The first level is the acquisition. The second level is the build and development. The third level is the deployment. The fourth level is the run command. The final level is the maintenance [7]. The platform used for managing containers is called as Kubernetes. This platform is used for providing effective communication between the containers. Further, it provides necessary resources to operate the containers in an independent manner. Using Kubernetes the reliability of the containers enhances to a greater extent [8]. The main drawback of these systems is the vulnerability to be attacked easily. This is because docker platform can be easily exposed to the external resources. Denial of service attack is the common attack that affects these docker images. Hence, preservation of docker images is a crucial task [9]. To integrate the cloud computing with the end users fog computing techniques are being used. These techniques increase the storage and resources of the cloud data center. This helps in the improved performance of the Internet of Things (IoT) devices. The fog architecture is usually split into multiple presence-of-points to achieve improved performance [10].

Section 2 provides methods, it contains related work research and a review of the literature. Section 3 provides the paper contributions. Section 4 provides the stages of the container maturity process. Section 5 describes lightweight container technologies and their management. Section 6 explains the implementation of the container on the host machine. Sections 7 describe the container load balancing with the proposed algorithm. Section 8 shows the result and discussions. Finally, Section 9 shows the conclusion of the work and the scope for future work.

2 Method

This section provides related theories and technologies that are the tools integrated into our scalable container service architecture.

Luo et al. [11] proposed a new fog computing architecture based on container systems for the energy balancing. The scheduling was done using improve the battery efficiency of the virtual machine. The resource utilization done by the fog devices are increased using the proposed technique. This system was designed to reduce the service delay. Von Leon et al. [12] designed a lightweight container model that is suitable for the cloud architecture. This scheme utilized the integration of cloud and container devices to enable reliable connectivity along with reduced computational power. The flexibility of the service was

attained using multi-tenancy containerization model. Garg et al. [13] introduced automated cloud infrastructure for the creation of robust container security. This was implemented using continuous integration of the docker containers. The transformation of cloud computing for the purpose of industrial automation purpose was implemented to improve the productivity of the system. Bhimani et al. [14] analyzed the variation of the performance of the container system with respect to the increase in the number of applications. This scheme was dedicated for the usage in intensive applications. A platform of multiple docker containers were used in this model to identify the effects on the performance in terms of execution time and resource utilization. Mendki [15] utilized docker containers for the implementation by the IoT edge devices. Deep learning framework was implemented in this system. The hardware component used in implementation was the Raspberry Pi module. This scheme was evaluated using surveillance applications in which the real-time data was analyzed. Wan et al. [16] used micro services for the implementation of the docker containers. Hypervisor based virtualization scheme was introduced in this framework. The objective was to minimize the application deployment cost using a micro service architecture. The algorithm was designed to operate in an incremental manner. Jha et al. [17] used holistic evaluation for the integration of docker containers with the micro services. Comparison was done using the container with hypervisor and container with bare-metal. In this work, the docker containers were implemented using heterogeneous set of micro services. Two types of containers namely inter-container and intra-container were analyzed and evaluated. Liu et al. [18] proposed a new scheduling algorithm for the container systems based on the optimization of multiple objectives. The resource scheduling was done using the Multiopt algorithm. A new metric was designed to weight each factor in scheduling. Based on this metric, a scoring function was implemented to combine all the factors. Saha et al. [19] evaluated the docker containers using scientific work loads. This system was designed for the cloud implementation. This scheme enabled the flexibility and portability for the usage with multiple applications. Solution was provided based on the singularity principle. Further, this scheme was capable of providing improved performance with minimal overhead. Lingayat et al. [20] performed an extensive performance evaluation of docker containers on baremetals. Here, containers were used for initiating the applications with minimal start up time. Testing was conducted to analyze the type of environment to be used for the implementation of the docker systems. It was found that the usage of baremetal improved the performance by 50%. Based on the literature survey, we infer that proper scheduling algorithms helps to improve the performance of docker systems. Hence, a new algorithm called Ant Colony Optimization-based Light Weight Container (ACO-LWC) load balancing scheduling algorithm for scheduling various process requests is being proposed in this research.

3 Contribution of the Paper

The main contributions of the paper are listed below:

- Virtual machine and Container systems are compared in detail.
- Lightweight container technologies and their management are discussed.
- Implementation of docker systems is explained.
- A novel Ant Colony Optimization-based Light Weight Container (ACO-LWC) load balancing scheduling algorithm is proposed.
- The proposed algorithm is analyzed using parameters like response time (ms), mean square error (MSE), node load, largest TPS of cluster (fetches/sec), average response time for each request (ms) and run time (s).

4 Virtual Machine *vs.* Container

The popularity of software containers has been increasing significantly since the release of Docker in 2013. The main advantage of virtualization approach is its lightweight nature. This helps to isolate and

run various programs even on computers that have limited computing capabilities. Recently, the docker containers are being used on top of virtual machine. This helps to overcome the drawbacks of earlier virtual machines. With the rapid evolution of IoT systems, the docker containers are used to promote highly connected networks. These systems help to improve the software processing speed, platform independence and process reliability.

Tab. 1 shows the main differences between virtual machine and the docker container. The first difference is the virtualization level. The virtual machine is based on hardware level whereas, the docker container is based on software level. The virtual machine is fully isolated and the docker container is isolated based on process or application level. The docker container is faster compared to virtual machine. Also, it occupies less storage space compared to virtual machine. The docker machine used a docker hub for its operation. The code reusability concept is encouraged on docker container compared to virtual machine. Further, the resource utilization is done dynamically in docker container whereas, it is done in a static manner in virtual machine.

Table 1: Virtual machine *vs.* container comparison using various parameters

Parameters	Virtual machine	Docker container
Level virtualization	Hardware level	Software level
Isolation	Fully isolated	Process or application-level isolation
CPU processing and performance	Slower: uses more CPU cycles	Faster: consumes less CPU cycle
Memory usage speed	Slower	Real-time and faster
Hardware storage	Occupies more storage	Less storage
Web-hosted hub	No web hub	Use docker hub
Code-reusability	Poor code-reusability	Encourage code-reusability
Resource utilization	Static allocation	Dynamic allocation

Fig. 1 shows the layer structure of the docker container. The lowermost layer is the infrastructure layer. This layer is topped with Guest Operating System. On top of this layer, is the docker layer. The top most layer is the application layer that contains multiple containerized applications.

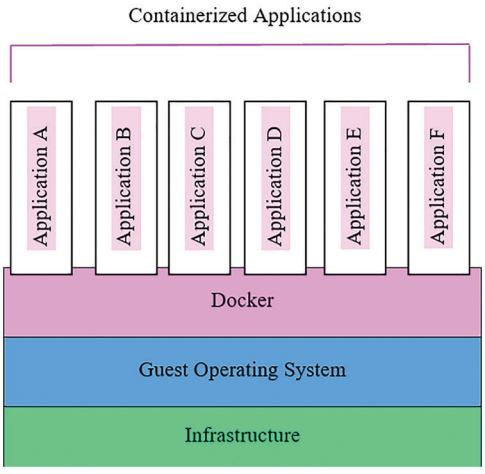

Figure 1: Layer structure of docker container

5 Lightweight Container Technologies and their Management

Container technology is a methodology that is capable of functioning as a packaged application. This technology operates with the help of its dependencies. Docker is a software container deployment tool. This tool operates at a process level. The docker container is a tool that allows applications to be isolated from one another in a lightweight manner. The container technology is a network-level virtualization technology that has a separate file system, network and process space to run a server without changing the host operating environment. Using a docker container, more than one processes can be operated simultaneously in a single computer without any conflict with each other.

Fig. 2 depicts the docker core concept. The main component of docker system is image. This image is a read-only model which is a combination of file system, configuration data and startup command. The docker containers have their registry on Docker Hub. Alternatively, they can also be operated on their private registry server. The Docker engine is responsible for starting, stopping and monitoring the containers on a given host. The popularly used CLI (command Line Interface) docker commands are as follows:

(i) docker pull: This command retrieves a resource from a dedicated repository, such as a docker image.
(ii) docker rmi: This command is used to delete one or more images from a container.
(iii) docker start: This command is used to start the container.
(iv) docker stop: This command terminates the operating container.
(v) docker commit: This command allows the user to build a new image based on changes made in an existing container.
(vi) docker ps: This command generates a list of containers.
(vii) docker rm: This command lets the user delete one or more containers.

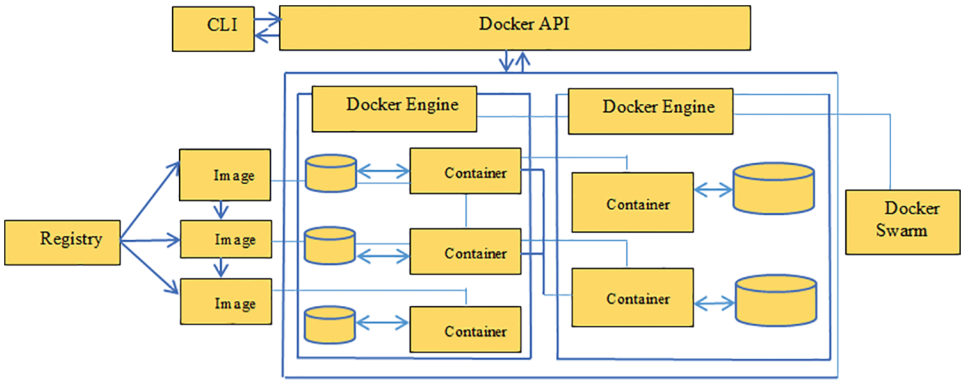

Figure 2: The core concept of docker container

6 Implementation

Implementation of docker with multi-service using load balancing algorithm is done as follows,

Step 1: A Dockerfile includes a collection of instructions for building an image of docker. The popularly used instructions are FROM, COPY and ENV. The FROM instruction is used for specifying the base images. The COPY command is used for the addition of new files into the container files. The ENV instruction is used for defining the default execution command for the environment variables.

Step 2: The second step is to build the image and to start the container based on the specification in docker-compose.yml file. In this step, the docker compose files are created. This is done using three main

steps. The first step is to define the application environment using Dockerfile. The second step is to define all the docker services using docker compose YAML file.

This file is shown in Fig. 3. The final step is to run the docker compose file using the command prompt.

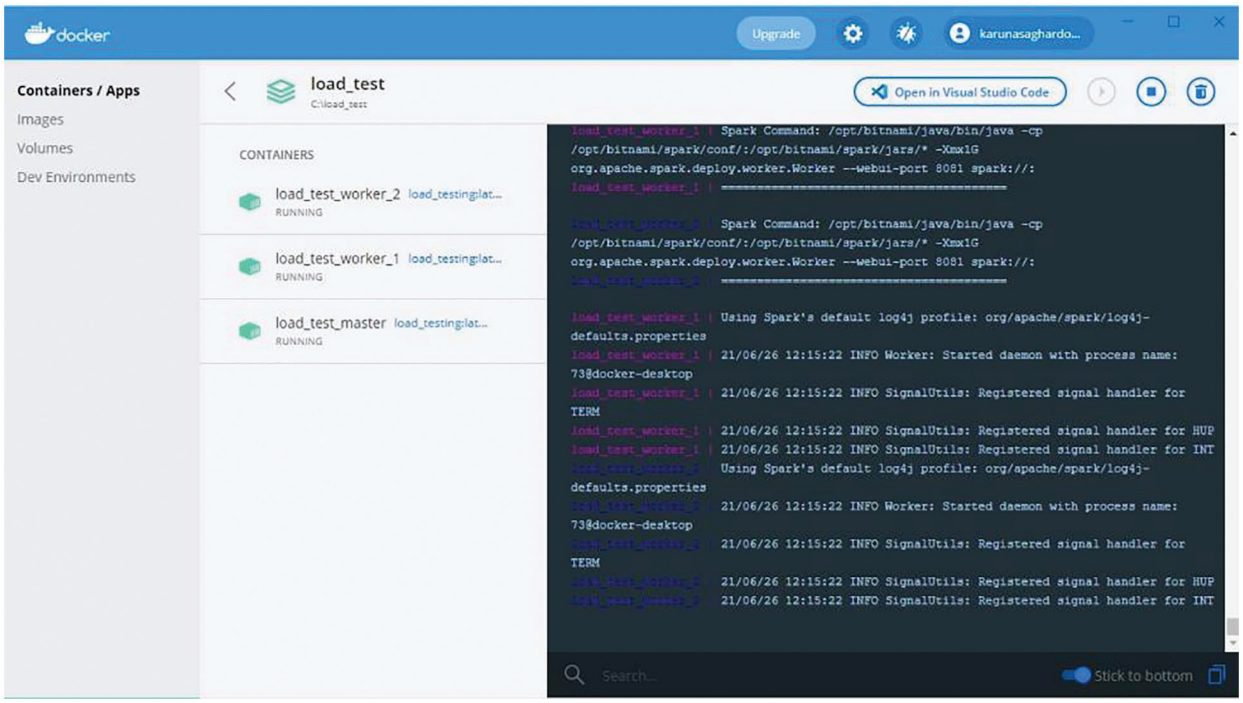

Figure 3: Docker-compose up

Step 3: The final step is to run the application using compose file. This step is used for building images from the Dockerfile. In this step the docker-compose up command is used for combining the container output.

Step 4: In this step, the docker stat command is used for monitoring the usage of the docker.

This is shown in Fig. 4. Here, the load_test_master is used for monitoring the service of the docker in a continuous manner.

CONTAINER ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET I/O	BLOCK I/O	PIDS
a5987e7f5362	load_test_master	0.14%	501.1MiB / 125.9GiB	0.39%	0B / 0B	184MB / 434kB	68
c050e81c0131	load_test_worker_2	161.83%	2.038GiB / 125.9GiB	1.62%	0B / 0B	3.64MB / 104MB	246
f2547940bd99	load_test_worker_1	385.81%	2.345GiB / 125.9GiB	1.86%	0B / 0B	38.1MB / 104MB	349
a5987e7f5362	load_test_master	0.14%	501.1MiB / 125.9GiB	0.39%	0B / 0B	184MB / 434kB	68
c050e81c0131	load_test_worker_2	559.71%	2.078GiB / 125.9GiB	1.65%	0B / 0B	3.64MB / 104MB	246
f2547940bd99	load_test_worker_1	392.90%	2.353GiB / 125.9GiB	1.87%	0B / 0B	38.1MB / 104MB	349
a5987e7f5362	load_test_master	0.13%	501.1MiB / 125.9GiB	0.39%	0B / 0B	184MB / 434kB	68
c050e81c0131	load_test_worker_2	559.71%	2.078GiB / 125.9GiB	1.65%	0B / 0B	3.64MB / 104MB	246
f2547940bd99	load_test_worker_1	392.90%	2.353GiB / 125.9GiB	1.87%	0B / 0B	38.1MB / 104MB	349
a5987e7f5362	load_test_master	0.13%	501.1MiB / 125.9GiB	0.39%	0B / 0B	184MB / 434kB	68

Figure 4: Docker stats

7 Proposed Load Balancing Scheduling Algorithm

A new algorithm is proposed for load balancing scheduling based on Ant Colony Optimization (ACO). In this scheduling algorithm, initially, the amount of resource to be allocated to each node is computed. Based on this computed value, the task allocation is done in a round robin fashion. Then the probability of transition from one node to another is computed. We mostly run docker on the bare computer. Where only docker is operating, implying that the system is completely dedicated to docker. Whereas the individual container will get a maximum of 60% CPU and memory consumption. For effective load balancing, the CPU usage and memory usage must be within a particular optimal range. Hence the CPU usage and memory usage are then computed. Finally, based on the probability of transition, rate of CPU usage and rate of memory usage, the next node to be selected by the artificial ant is then identified. This is explained in the Algorithm 1 below.

Algorithm 1: Ant Colony Optimization-based Light Weight Container (ACO-LWC) load balancing scheduling algorithm

The scheduler is designed such that the tasks are allocated based on the available resources. In this research, ACO is used for the optimal task allocation. The pheromone trail of every resource is analyzed by the artificial ant for the computation of amount of resource to be allocated to each node. This is given by,

$$R(N_i) = \omega_m * \left(\frac{M'(N_i)}{M(N_i)}\right) + \omega_c * \left(\frac{C'(N_i)}{C(N_i)}\right) \tag{1}$$

where $R(N_i)$ is the allocated resource for the node N_i, ω_m is the memory weight, $M'(N_i)$ is the amount of memory available in node N_i, $M(N_i)$ is the total memory of node N_i, ω_c is the CPU weight, $C'(N_i)$ is the available CPU resources in node N_i and $C(N_i)$ is the total CPU of node N_i.

Then, the task allocation for each node is done in a Round-Robin fashion. It is represented as

$$T_0(N_i) = RR(N_i) \tag{2}$$

where T_0 represents the initial task allocation and RR represents the Round-Robin algorithm.

The probability of transition from node N_i to node N_j at the T^{th} instant is then given by,

$$P_T(N_i,\ N_j) = \frac{T_o(N_i) * (\lambda_i)}{\sum_{j=1}^{n} T_o(N_j) * (\lambda_j)} \tag{3}$$

where $P_T(N_i, N_j)$ is the probability of transition from node N_i to node N_j, $T_o(N_i)$ is the task allocated to node N_i, λ_i is the heuristic value of node N_i, $T_o(N_j)$ is the task allocated to node N_j, λ_j is the heuristic value of node N_j and n represents the total number of nodes.

The rate of CPU usage is computed as

$$\sigma_C = \frac{C^U(N_i)}{C(N_i)} \times 100 \tag{4}$$

where σ_C is the rate of CPU usage, $C^U(N_i)$ is the amount of CPU used by node N_i and $C(N_i)$ is the total CPU of node N_i.

The rate of memory usage is computed as

$$\sigma_M = \frac{M^U(N_i)}{M(N_i)} \times 100 \tag{5}$$

where σ_M is the rate of memory usage, $M^U(N_i)$ is the amount of memory used by node N_i and $M(N_i)$ is the total memory of node N_i.

If the CPU usage level is too high, then the response time increases. If the CPU usage level is too low, then the CPU resource gets wasted. Thus, the CPU usage level must be within certain limits. Similarly, if the memory usage level is too high, the run time increases. If the memory usage level is too low, the node load gets unbalanced. Thus, the memory level should neither be too low nor be too high. Based on the probability of transition, rate of CPU usage and rate of memory usage, the next node to be selected by the artificial ant, i.e., node N_j is identified using,

$$N_j = \begin{cases} \arg\max_{j \in n} P_T(N_i, N_j); & \sigma_{LC} < \sigma_C < \sigma_{HC} \ \& \ \sigma_{LM} < \sigma_M < \sigma_{HM} \\ P_T(N_i, N_j) & ; o.w. \end{cases} \tag{6}$$

where σ_{LC} is the lower limit of the CPU usage rate, σ_{HC} is the upper limit of the CPU usage rate, σ_C is the rate of CPU usage, σ_{LM} is the lower limit of the memory usage rate, σ_{HM} is the upper limit of the memory usage rate and σ_M is the rate of memory usage. The value of σ_{LC}, σ_{HC}, σ_{LM} and σ_{HM} are set as 0.2, 0.8, 0.2 and 0.8 respectively.

The lightweight container load-balancing method is based on ant colony optimization. Fig. 5 shows the flow chart of the proposed ACO-LWC Load Balancing Algorithm. The flowchart goes as follows: first, initialize the container parameters, such as the container name and then begin generating a Docker application package. Then set CPU and memory threshold limitations. The suggested ACO-LWC Load Balancing Algorithm is used to verify the container's workload. If the workload of the container reaches the maximum threshold limit, ant proceeds to compute the container's current resource consumption. If the container is found to be overloaded, a new container is created and a job is assigned. Finally, the best solution was discovered.

8 Results and Discussion

For evaluating the proposed ACO-LWC algorithm we have used parameters like response time (ms), mean square error (MSE), node load, largest TPS of cluster (fetches/sec), average response time for each request (ms) and run time (s). The proposed algorithm is compared with two existing load balancing scheduling algorithms namely, least connection algorithm and round robin algorithm.

In Fig. 6, the percentage of requests processed were varied from 50% to 95% and the response time was identified for the three algorithms namely, least connection, round robin and the proposed ACO-LWC algorithm. From the Fig. 6, we see that the response time is minimal for the proposed ACO-LWC algorithm compared to the existing algorithms. The response time for least connection, round robin and the proposed ACO-LWC algorithm are 25, 28 and 18 ms respectively when 50% requests are finished. This shows that the response speed is maximum for the proposed scheme. Thus, this scheme can be easily implemented in docker containers for efficient usage. Also, Fig. 5 shows that the response time for least connection, round robin and the proposed ACO-LWC algorithm are 58, 60 and 48 ms respectively when 95% requests are finished. This shows the effectiveness of the proposed algorithm.

Fig. 7 shows the comparison of mean square error of the three scheduling algorithms, after scheduling 60 and 120 requests. For all the three algorithms, the error increases when the number of requests increases. The error for scheduling 60 requests using least connection, round robin and the proposed ACO-LWC algorithm are 0.12, 0.09 and 0.04 respectively. Similarly, the error for scheduling 120 requests using least connection, round robin and the proposed ACO-LWC algorithm are 0.15, 0.11 and 0.06 respectively. Thereby, it is evident that the error is very less using the proposed load balancing algorithm. This indicates that the proposed algorithm achieves better performance in load balancing with better stable results.

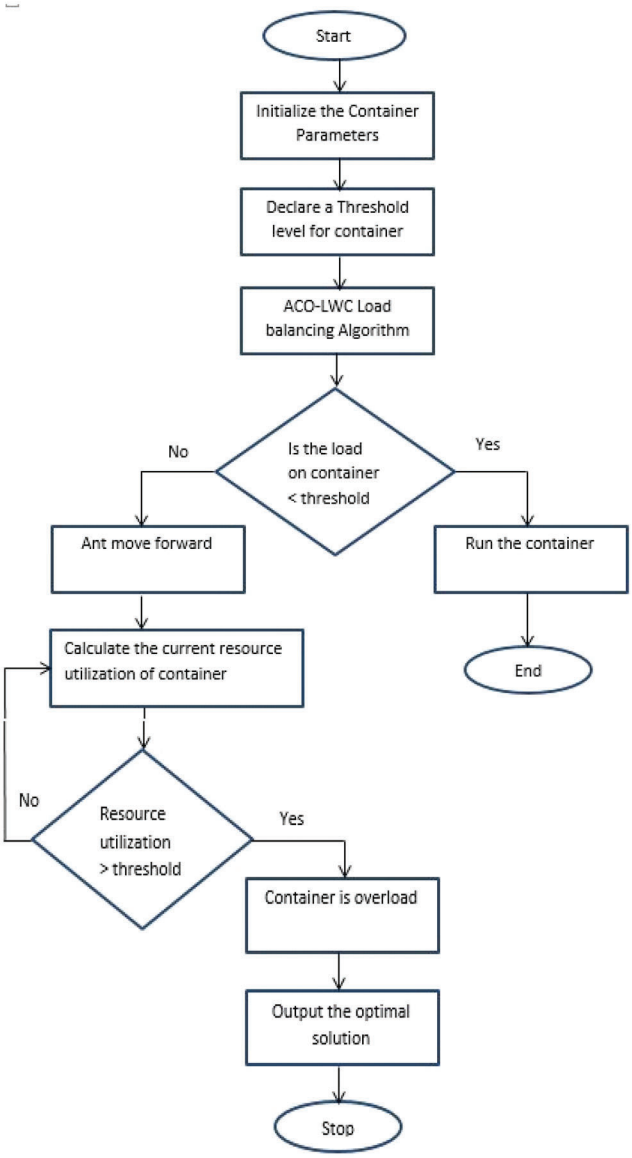

Figure 5: Flow chart for the proposed ACO-LWC load balancing algorithm

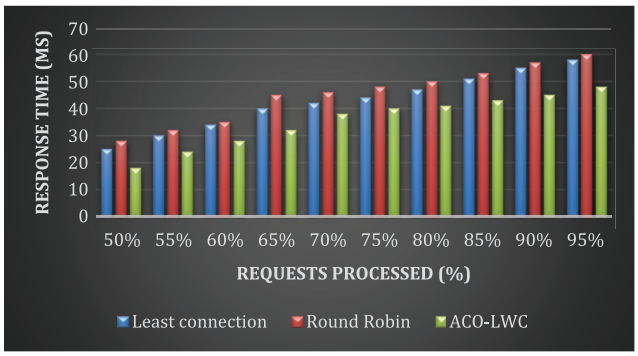

Figure 6: Comparison of response time

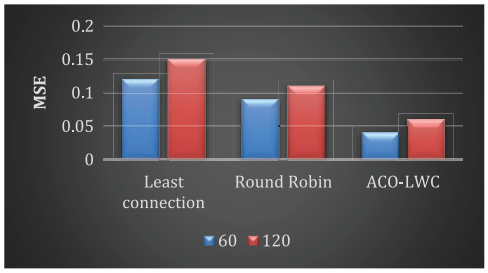

Figure 7: Comparison of mean square error

The cluster is divided into many "clusters" in the sense that each node can continue to ping just a subset of the nodes it is aware of. The maximum number of compute nodes in a cluster network is 128 nodes. In Fig. 8, we have considered three nodes namely, Node 1, Node 2 and Node 3. We find that the node load cluster is dispersed in the least connection and round robin algorithm. Whereas, for the proposed ACO-LWC algorithm, the node load is closer together in the cluster. The closeness of node load in the cluster indicates the stability of load balancing. Thus, we can infer that the proposed ACO-LWC algorithm achieves better stability performance.

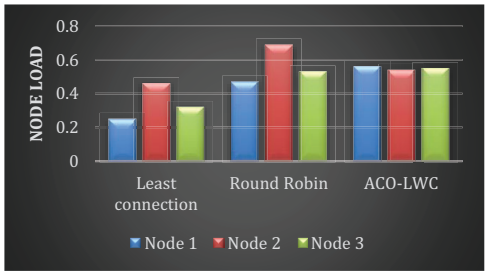

Figure 8: Comparison of node load

Fig. 9 shows the variation of largest TPS of cluster with respect to number of applications. As shown in the Fig. 9, we see that the number of TPS increases tremendously using the proposed ACO-LWC algorithm. However, the TPS for the least connection and round robin are very low even for higher number of applications. The TPS of cluster for 20 applications is 45, 55 and 650 for the least connection, round robin and proposed ACO-LWC respectively. Similarly, the TPS of cluster for 50 applications is 120, 231 and 1042 for the least connection, round robin and proposed ACO-LWC respectively. The reason for this is because the proposed algorithm schedules the containers adaptively based on the CPU and memory usage. Thus, the response speed is tremendously increased that results in improved TPS performance.

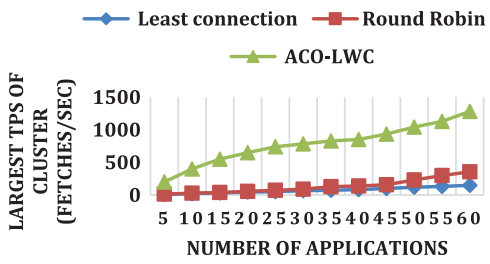

Figure 9: Variation of largest TPS of cluster with respect to number of applications

In Fig. 10, we see that, for the least connection algorithm, as the number of applications increases, the average response time also increases. Thus, this system is not suitable for higher number of applications. The TPS range achieved by the round robin algorithm is in the range of 2345 to 2865 ms. Thus, it is clear that the response time is high for the round robin algorithm. The proposed ACO-LWC algorithm attains minimal response time ranging from 1567 to 1593 ms. For higher number of applications, for instance, when the number of applications is fixed as 60, algorithms like least connection, round robin and the proposed scheme uses a response time of 5282, 2865 and 1593 ms. Thereby, we understand that, even when the number of applications is high, the response time of the proposed scheme is minimal. This aids in the easier implementation of the proposed algorithm.

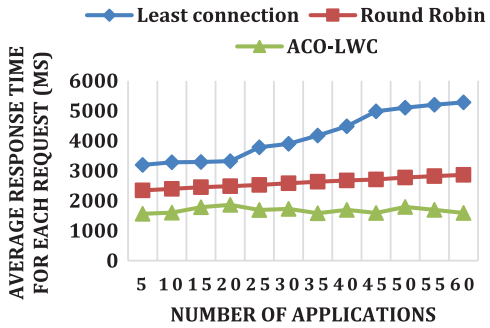

Figure 10: Variation of average response time for each request with respect to number of applications

In Fig. 11 see that, the run time increases with the increase in the number of applications for all the three algorithms. However, the average run time (ART) for the proposed ACO-LWC scheme is 42.08 ms. For least connection algorithm, the ART is 87.75 ms. For the round robin algorithm, the ART is 67.66 ms. This shows that the overall run time is very low for the proposed scheme. There is around 2.08 times reduction in ART compared to least connection and 1.607 times reduction in ART compared to the round robin algorithm. This clearly indicates the increased performance speed of the proposed scheme.

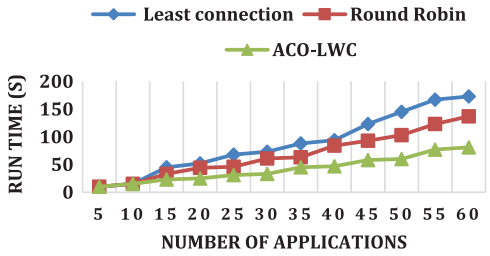

Figure 11: Variation of run time with respect to number of applications

8.1 CPU Performance

The performance of the CPU is greatly improved by the utilization of the proposed ACO-LWC load balancing scheduling algorithm. This helps in the effective resource utilization of the docker host systems. Further, the time taken for the completion of a particular task is very less with the usage of the proposed scheme.

Fig. 12 clearly indicates that the proposed scheme has a higher capability to limit the CPU utilization. The amount of CPU utilized by containers and images are greatly reduced with the usage of the proposed algorithm.

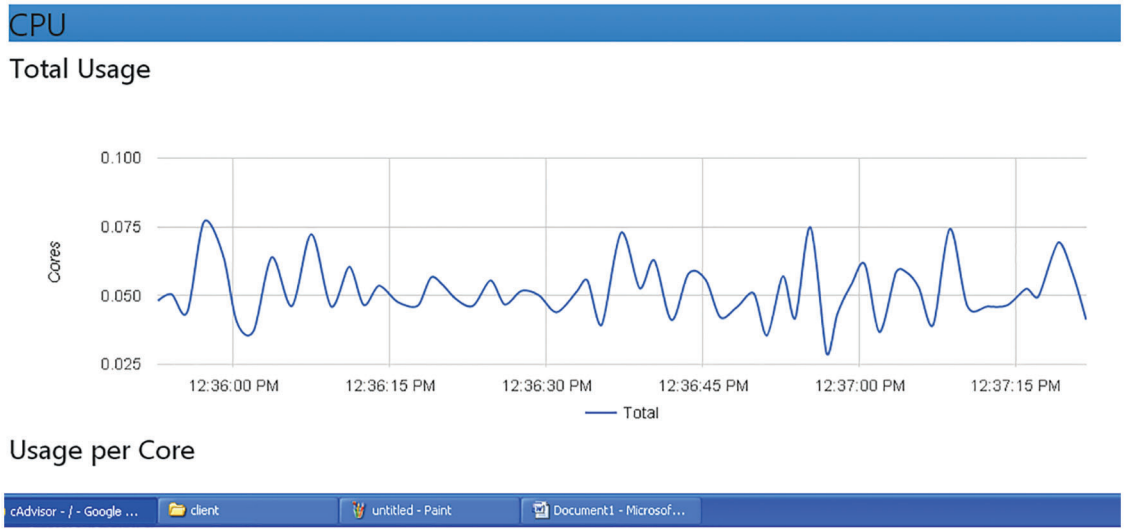

Figure 12: CPU utilization of ACO-LWC scheme

Tab. 2 shows the comparison of CPU usage (%) for all the three algorithms. The Tab. 2 indicates that the CPU usage for least connection, round robin and ACO-LWC algorithm are 0.43, 0.27 and 0.10 respectively. This shows that the proposed algorithm has very less CPU usage, thereby, maximizes the CPU efficiency.

Table 2: Comparison of CPU usage

Algorithm	CPU usage (%)
Least connection	0.43
Round robin	0.27
ACO-LWC	0.10

8.2 Memory Performance

The proposed scheme is further analyzed with respect to the memory performance. This indicates the easier usage of the computer resources. Efficient memory performance helps the docker systems to process multiple applications with minimal cost and minimal overhead.

Fig. 13 shows the RAM speed performance of the proposed ACO-LWC algorithm. This Fig. 13 clearly illustrates that almost 100% of the memory utilized by the system are allocated to the software. This ensures that very low memory level is utilized by the system for the virtualization. This helps to improve the RAM speed and the overall performance of the system.

The Tab. 3 indicates that the memory usage for least connection, round robin and ACO-LWC algorithm are 2.13 GB, 1.53 GB and 144.3 MB respectively. Thus, the memory utilization of the proposed algorithm is very low. This increases the RAM speed of the system.

8.3 Network I/O Performance

The network I/O indicates the data quantity transmitted and received by the docker system using the network interface. The proposed scheme is analyzed based on the network I/O levels using the throughput of the system.

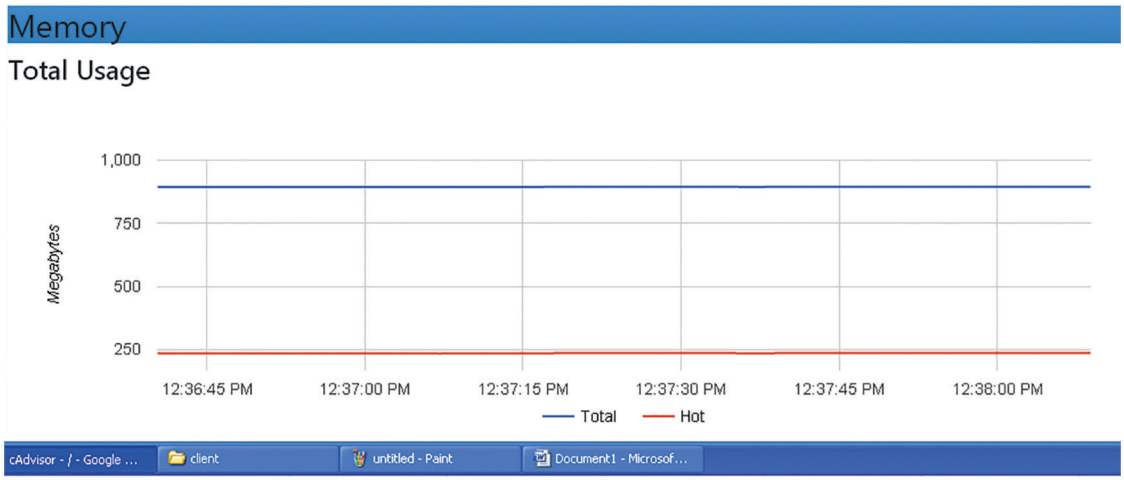

Figure 13: RAM speed performance of ACO-LWC scheme

Table 3: Comparison of memory usage

Algorithm	Memory usage
Least connection	2.13 GB
Round robin	1.53 GB
ACO-LWC	144.3 MB

Fig. 14 illustrates the network I/O performance of the proposed ACO-LWC scheme. This Fig. 14 indicates that the performance of the system in terms of throughput is very high. Also, high throughput is achieved even in the presence of under loading and overloading conditions. This shows the effective load balancing capability of the proposed scheme.

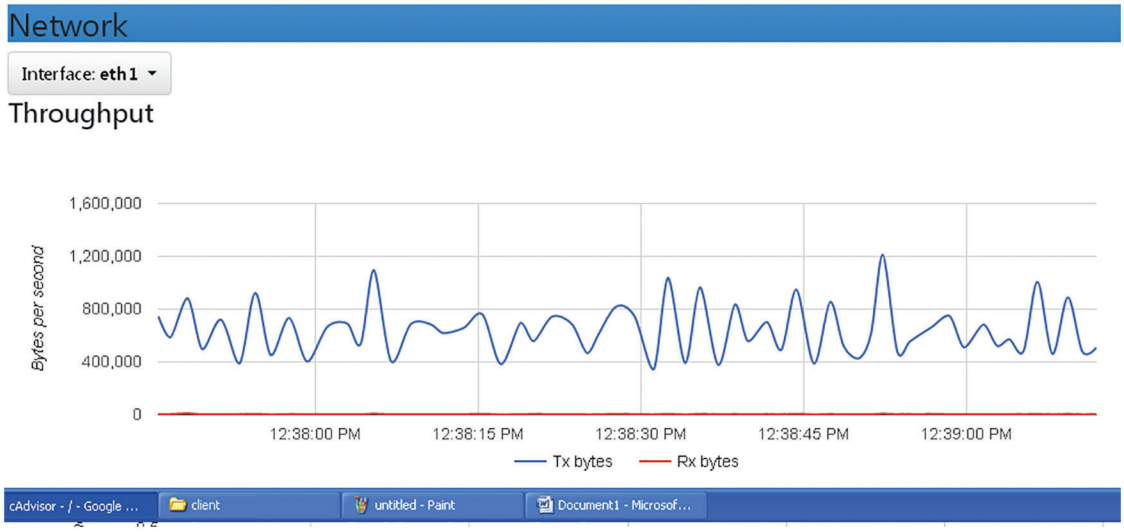

Figure 14: Network I/O performance of the proposed ACO-LWC scheme

9 Conclusion and Future Work

In this research, we have proposed a new algorithm called Ant Colony Optimization-based Light Weight Container (ACO-LWC) load balancing scheduling algorithm for scheduling various process requests. Initially, the task allocation was done in a round robin fashion. To achieve effective load balancing, in this algorithm, the CPU usage and memory usage were maintained within a particular optimal range. Here, the load balancing scheduling was done based on Ant Colony Optimization. Performance analysis showed that the proposed ACO-LWC algorithm achieved better stability performance. Further, the TPS of cluster for 50 applications was found to be 120, 231 and 1042 for the least connection, round robin and proposed ACO-LWC respectively. Similarly, the response time for least connection, round robin and the proposed scheme with 60 applications was identified as 5282, 2865 and 1593 ms. Furthermore, analysis shows that the overall run time is very low for the proposed scheme. There is around 2.08 times reduction in run time compared to least connection and 1.607 times reduction in run time compared to the round robin algorithm. In future, we plan to integrate the proposed framework with IoT systems. We also plan to deploy the proposed algorithm for micro services-based applications.

Acknowledgement: The author thanks the institution and stakeholders for their support and the reviewers for their insights into the paper, because these comments have led to improved work.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] A. M. Potdar, D. G. Narayan, S. Kengond and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2019.

[2] K. Brady, S. Moon, T. Nguyen and J. Coffman, "Docker container security in cloud computing," in *10th Annual Computing and Communication Workshop and Conf.*, Las vegas, NV, USA, pp. 975–980, 2020.

[3] A. Zerouali, T. Mens, G. Robles and J. M. Gonzalez-Barahona, "On the relation between outdated docker containers, severity vulnerabilities, and bugs," in *Proc. of the IEEE 26th Int. Conf. on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, pp. 491–501, 2019.

[4] M. Lin, J. Xi, W. Bai and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83088–83100, 2019.

[5] M. Sollfrank, F. Loch, S. Denteneer and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3566–3576, 2021.

[6] S. Kwon and J. H. Lee, "DIVDS: Docker image vulnerability diagnostic system," *IEEE Access*, vol. 8, pp. 42666–42673, 2020.

[7] E. Casalicchio and S. Iannucci, "The state-of-the-art in container technologies: Application, orchestration and security," *Concurrency and Computation Practice and Experience*, vol. 32, no. 17, pp. 1–17, 2020.

[8] J. Shah and D. Dubaria, "Building modern clouds: Using docker, kubernetes google cloud platform," in *Proc. of the 9th IEEE Annual Computing and Communication Workshop and Conf. (CCWC)*, Las vegas, NV, USA, pp. 184–189, 2019.

[9] T. Watts, R. G. Benton, W. B. Glisson and J. Shropshire, "Insight from a docker container introspection," in *Proc. of the Annual Hawaii Int. Conf. on System Sciences*, Grand Wailea, Hawaii, pp. 7194–7203, 2019.

[10] A. Ahmed, A. Mohan, G. Cooperman and G. Pierre, "Docker container deployment in distributed Fog infrastructures with checkpoint/restart," in *Proc. of the 8th IEEE Int. Conf. on Mobile Cloud Computing, Services and Engineering*, Oxford, United Kingdom, pp. 55–62, 2020.

[11] J. Luo, L. Yin, J. Hu, C. Wang, X. Liu *et al.,* "Container-based fog computing architecture and energy-balancing scheduling algorithm for energy IoT," *Future Generation Computer Systems*, vol. 97, pp. 50–60, 2019.

[12] D. Von Leon, L. Miori, J. Sanin, N. El Ioini, S. Helmer *et al.,* "A lightweight container middleware for edge cloud architectures," *Fog Edge Computing: Principles and Paradigms*, 7, pp. 145–170, 2019.

[13] S. Garg and S. Garg, "Automated cloud infrastructure, continuous integration and continuous delivery using docker with robust container security," in *Proc. of the 2nd Int. Conf. Multimedia Information Processing and Retrieval (MIPR)*, USA, pp. 467–470, 2019.

[14] J. Bhimani, Z. Yang, N. Mi, J. Yang, Q. Xu *et al.,* "Docker container scheduler for I/O intensive applications running on NVMe SSDs," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 313–326, 2018.

[15] P. Mendki, "Docker container based analytics at IoT edge video analytics usecase," in *Proc. of the 3rd Int. Conf. on Internet of Things: Smart Innovation and Usages (IoT-SIU)*, Nainital, Uttarakhand, India, pp. 1–4, 2018.

[16] X. Wan, X. Guan, T. Wang, G. Bai and B. Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *Journal of Network and Computer Applications*, vol. 119, no. May, pp. 97–109, 2018.

[17] D. N. Jha, S. Garg, P. P. Jayaraman, R. Buyya, Z. Li *et al.,* "A holistic evaluation of docker containers for interfering microservices," in *Proc. of the IEEE Int. Conf. on Services Computing (SCC)*, no. VM, pp. 33–40, 2018.

[18] B. Liu, P. Li, W. Lin, N. Shu, Y. Li *et al.,* "A new container scheduling algorithm based on multi-objective optimization," *Soft Computing*, vol. 22, no. 23, pp. 7741–7752, 2018.

[19] P. Saha, P. Uminski, A. Beltre and M. Govindaraju, "Evaluation of docker containers for scientific workloads in the cloud," in *ACM Int. Conf. Proc. Series*, USA, pp. 1–8, 2018.

[20] A. Lingayat, R. R. Badre and A. K. Gupta, "Performance evaluation for deploying docker containers on baremetal and virtual machine," in *Proc. of the 3rd Int. Conf. on Communication and Electronics Systems (ICCES)*, Coimbatore, India, pp. 1019–1023, 2018.