International Journal of Cloud Applications and Computing
Volume 13 • Issue 1

Resource-Aware Least Busy (RALB) Strategy for Load Balancing in Containerized Cloud Systems

Zakariyae Bouflous, ENSEM, Hassan 2 University of Casablanca, Morocco*
https://orcid.org/0009-0006-4200-7341

Mohammed Ouzzif, EST, Hassan 2 University of Casablanca, Morocco

Khalid Bouragba, EST, Hassan 2 University of Casablanca, Morocco

ABSTRACT

Containers are a key technology in modern cloud environments. They provide a lightweight manner to package and deploy service applications compared to virtual machines, with an isolated processing approach. The state-of-the-art containers LB algorithms do not take into account efficient mapping between container requirements and servers' capabilities, resulting in wastage of resources and increased response time, which negatively impacts the overall QoS. This manuscript proposes a new LB algorithm for containers called resource-aware least busy (RALB). RALB not only considers the current load of each node (as per the least connection) but also takes into account the current resource requirements of containers. It assigns containers to the least busy server with optimal capacity from a resource perspective. The authors implemented a prototype of the algorithm, and simulation results shows that more even distribution of workload and better utilization of resources across the nodes can be achieved, resulting in lower latency and high reliability compared to the randomized algorithm.

KEYWORDS

Capacity-Based Scheduling, Cloud Computing, Containerized Cloud, Distributed Systems, Load Balancing, Resource-Awareness

INTRODUCTION

In the realm of cloud computing (CC) architectural design, virtualization serves as a fundamental principle, facilitating the transition from on-premise infrastructure to a scalable model of remotely connected services. The isolation of machines and applications is crucial to ensuring the delivery of computing resources through an on-demand network infrastructure. Traditionally, a hypervisor is responsible for creating and managing virtual machines (VMs) that host applications and handle client requests. This approach is primarily suitable for conventional monolithic architecture-based applications, where all software components are designed, built, deployed, and managed as a single

DOI: 10.4018/IJCAC.328094

*Corresponding Author

unit codebase (Guo & Yao, 2018). However, a recent paradigm shift has occurred with the introduction of container-based microservices. This approach allows applications to be allocated on multiple lightweight containers, distributed across a large number of nodes (Thongthavorn & Rattanatamrong, 2019). Regardless of whether deployment is VM-based or container-based, cloud services providers (CSP) are expected to ensure system availability and throughput for different customers under a contracted service level agreement (SLA) acceptance criterion. Load balancing (LB) algorithms play a vital role in guaranteeing quality of service (QoS) metrics and maximum availability of the cloud platform. The primary objective is to prevent any server from becoming overloaded or underloaded. LB algorithms aim to distribute tasks as evenly as possible across a set of nodes or computing resources (Sahana & Mukherjee, 2020), thereby enhancing business efficiency and reducing the overall makespan of the CC system. During the process, priority of each incoming task and its characteristics are taken in consideration (Thongthavorn & Rattanatamrong, 2019). When considering LB techniques, it is essential to address two key steps: (1) resource allocation and (2) VM/container live migration. Resource allocation involves the fair distribution of jobs in the queue to servers, VMs, or containers based on the system's current state and the hardware capability of the hosts. This step ensures that jobs to resources are allocated efficiently and effectively. The task scheduling process is considered NP-hard problem where state-of-the-art applied strategies need to be improved (Aliyu et al., 2020). On the other hand, VM/container live migration enables the real-time transfer of application execution contexts, either from one physical host to another or within the same host, without disrupting the user experience. Several LB algorithms have been proposed in the literature for the same purposes, addressing both use cases of (VM) and container-based cloud systems.

Our Contribution

In this paper, our focus is on the container-based approach. We introduced a novel load balancing algorithm called Resource-Aware Least Busy (RALB), which aims to efficiently allocate and distribute container workloads among servers in the load balancing process. The "Resource Aware" keyword incorporates real-time monitoring of server's resource usage (CPU, memory, and bandwidth) before proceeding to any load balancing decision. The term "least busy" highlights the algorithm's core principle of allocating containers to servers with the lowest workload from a resource perspective. By considering server resources and allocating containers to the least busy server capable of meeting the container's requirements (CPU, memory, and bandwidth), RALB optimizes resource utilization and achieves effective load balancing. This approach sets it apart from the "least connection" algorithm, which predominantly considers the number of active connections. The rest of paper is organized as follows: Firstly, we will underline the container solution compared to standard virtualization with highlight to related LB works in containerization, then follow with a section describing the system model of the proposed algorithm. Subsequently, we will present the experimental results for the RALB alongside with comparison charts. Finally, we conclude our work and provide directions for improvements in the last section.

MOTIVATION AND RELATED WORK

Containerized vs. Virtualized Cloud

Containerized cloud systems present a new model for designing cloud solutions, using packaging and isolation mechanisms to encapsulate services along with their dependencies into atomic-contained units called containers. Docker and Kubernetes are widely adopted technologies for development, deployment, and load balancing of containerized applications (Nguyen & Kim, 2020, pp. 2-3). Docker, an open-source platform, simplifies the process of building and deploying applications within isolated containers. In contrast, Kubernetes enables the orchestration of container runtime systems across a cluster of networked resources. The workflow typically involves creating and deploying containers using Docker in the initial stage, followed by managing the runtime, scaling, and overall lifecycle of these containers using Kubernetes. Figure 1 provides a visual representation of the container states or lifecycle within both technologies.

Compared to virtual machines, containers offer a more lightweight approach to virtualization, as they operate at the operating system level, while virtual machines virtualize at the hardware level (Tao et al., 2019). Another distinguishing factor is the type of isolation achieved. In the case of virtualization, isolation is achieved by creating separate virtual machines (VM1, VM2…VMn) using a hypervisor, each with its own guest operating system (OS). On the other hand, containers achieve isolation at the process level, with multiple containers (C1, C2…..Cn) running in the same environment sharing common resources (memory, CPU, and storage). In this scenario, the security and isolation of applications rely on the host operating system's security mechanisms and the underlying containerization technology. Figure 2 describes abstraction levels for both VM and container-based cloud architectures.

Overall, there are several key differences between virtual machine (VM) and container-based cloud systems:

- **Resource utilization:** Virtual machines run on a full copy of a guest operating system, resulting in heavier resource requirements. Containers, by contrast, share the same host operating system and only utilize the necessary resources to run the application and its dependencies.

Figure 1. The container states in docker and kubernetes

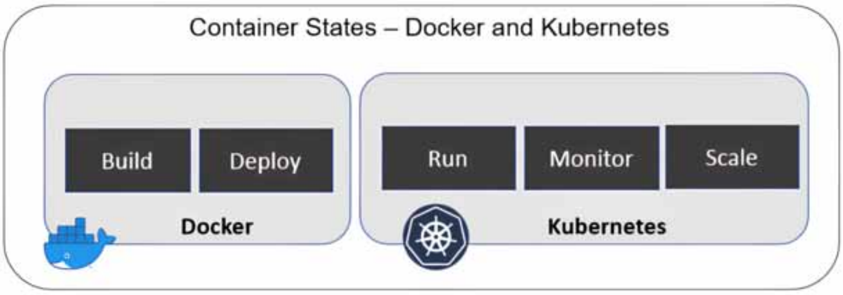
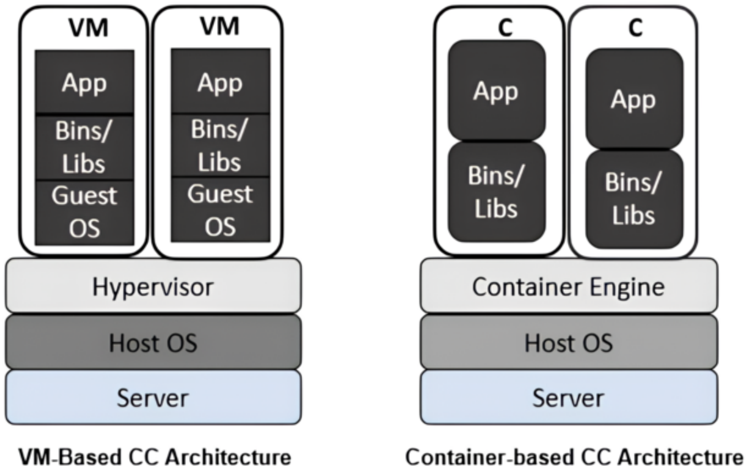


Figure 2. VM vs. container abstraction levels



- **Startup time:** VM typically takes several minutes to get started, while containers can be initiated almost instantly within a few seconds (Zhao et al., 2017).
- **Security:** VMs provide a higher level of security as virtualization occurs at the hardware level, ensuring isolation between the guest operating system and other VMs at the same host, whereas containers share the host operating system and rely on it to manage security and priorities for container execution contexts.
- **Portability:** Containers are more portable compared to VMs. They are lightweight and do not include entire OS binaries and libraries except necessary modules for their runtime execution. This makes container migration easier across different environments, including different cloud providers and on-premise data centers.
- **Management overhead:** Containers have less management overhead since they share the same operating system.
- **Live migration:** Live migration of containers is more complicated compared to VMs. Docker-based containers, with their layered image and shared kernel for scheduling introduce additional considerations for migration (Xu et al., 2020).

Related Work

Several load balancing algorithms have been proposed in the literature for both the containerized and VM-based cloud. We applied a rigorous selection process to filter out similar work and ensure the distinctiveness and novelty of our research. Our criteria included a comprehensive review of the literature to identify existing load balancing algorithms in containerized cloud environments. We specifically focused on algorithms that addressed resource-awareness, workload distribution, scalability, or energy-consumption. Publications were evaluated based on their relevance to our research objectives and their contribution in terms of novel methodologies or performance improvements in the area of workload management in the containerized clouds.

Sureshkumar and Rajesh (2017) proposed an energy-aware processing model to balance load on docker containers with energy reduction. They provided a solution to concentrate the workload on a subset of docker containers, replicate a new container whenever required for overloading use cases following threshold conditions, and then destroy a container when load is decreased by docker API. This solution optimizes energy consumption, since the smallest set of servers operates evenly at optimal energy. Based on load requirements, a state machine between sleep mode and active mode addresses energy utilization criterion for all containers, which improves overall power consumption for the cloud infrastructure.

Guo and Yao (2018) designed a container scheduling strategy based on the neighborhood division (CSBND) algorithm, which considers system load balancing and response time to improve overall system performance in micro service scenarios, differently to VM-based traditional load balancing algorithms. The idea is to consider call dependencies between micro service containers, which are allocated to different physical machines (PMs). A weighted graph is used first to build the container network cells, then a method of container neighborhood division is performed which is mandatory for the processing of the LB algorithm. Optimally, containers and their dependencies are more likely to be placed in neighboring hosts. Validation of the algorithm has been performed compared to MOPSO and spread algorithms considering workload required. CSBND shows a lower CPU and memory usage and a reduced network call among different PMs, with optimization of system performance up to 25%.

Kaur et al. (2019) addressed the optimization of energy utilization as well for data centers (DCs). They formulated a multi-objective optimization problem (MOOP) for container placement with the aim of balancing workload and minimizing overall energy consumption. Considering that 416.2 billion KWh of energy was utilized by DCs in 2016, CPU cores, bandwidth, and storage container requirements have been considered as attributes for the algorithm. The goal is to attain optimal efficient energy utilization while mapping containers to physical hosts. MOOP operates based on a master and host, called agents, which monitor energy consumption and workload balancing considering SLA. Simulations have been performed comparing MOOP to the first fit decreasing (FFD) approach, and experimental results showed that MOOP performs better than FDD in terms of energy utilization and SLA violation, at compromise of 10.39% higher load additionally on the hosts.

Patel et al. (2020) introduced a grey wolf optimization (GWO) algorithm for load balancing addressing the overall makespan of the cloud system. Their system model consists of a task manager and task scheduler main components. Containers' specifications have been investigated as a set of memory size (amount of RAM in GB) and processing capacity (MIPS). As metrics to determine the overall makespan of all tasks allocated to containers considering waiting time WT, execution time ET and completion time CT. Additionally, the algorithm also considers the load balance requirements for each container by introducing the load variation LV criterion. GWO has been analyzed in comparison to the genetic algorithm and particle swarm optimization (PSO) based algorithms. The experimental results indicate that GWO is more effective than the other algorithms in terms of load balancing and reducing makespan.

Patra et al. (2020) proposed a randomized algorithm for containerization in cloud environments. The algorithm considers jobs as balls and hosts as bins. The idea of the algorithm is split in two phases: 1) construction of a minimum edge weight graph with fully unidirectional connection of all servers and 2) allocation of balls/tasks into servers with minimum load via local search in neighboring servers. The edge weight or distance between two servers has been generated randomly in the first phase to make the graph fully connected. Experiments have been performed in four uses cases considering different numbers of tasks, and results showed that load balancing is performed smoothly.

Lin et al. (2019) proposed an ant colony algorithm for multi-objective optimization in microservices used for container-based cloud systems. They considered in their model the resource requirements of servers, the number of requests, and the failure rate of the physical nodes. It uses the quality evaluation function of feasible solutions to ensure the validity of pheromone updating and combines multi-objective heuristic information to improve the selection probability of the optimal path. First, network transmission overhead is calculated among all containers between CSP and CSC of the microservice. Second, the maximum resource utilization rate is calculated considering the MRLB optimization problem. Last, the average number of failures for a microservice request is determined. Experimental results showed that the proposed optimization algorithm achieves better results in the optimization of cluster service reliability, cluster load balancing, and network transmission overhead.

THE SYSTEM MODEL

This section illustrates the system framework of our proposed RALB algorithm. Considering a total of N servers, denoted by the set S= {S1, S2, . . . Sn, where (Si) refers to the ith node in the reference cloud data center, the variables cpu_usage, mem_usage, and bw_usage present the respective resource (cpu, memory, and bandwidth) usage for each server (Si) hosting a set of containerized applications. Correspondingly, cpu_cap, mem_cap, and bw_cap denote the maximum amount of resources a single node can process. The capacity of each server is utilized by RALB algorithm as a reference point for determining which node is currently the "least busy" from a resource perspective and therefore best suited for scheduling new containers. Each server (Si) is assumed to host a number x Î {1..M} of containers with their known respective resource requirements. Similarly, the notation C refers to the global set of containers C={C1, C2, . . . Cm} that are being allocated or migrated. The allocation of the container (Cj) to the server (Si) is denoted Cji. Each Container C_{ji}, is specified by its runtime resource requirements $C(ji)_{Cpu}$, $C(ji)_{Mem}$ and $C(ji)_{Bw}$. The system model of RALB consists of two main components:

- **Resource manager (RM):** This is responsible for supervising and monitoring the resource usage (RU) of each physical machine and submitting the outcome to the container scheduler. RU(Si) denotes the current resource usage for a server (Si) and the functional algorithm is specified in the following equation:

$$RU(Si) = \sum_{j=1}^{M} \frac{\left(Cji\right) cpu}{S(i)CpuCap} + \sum_{j=1}^{M} \frac{\left(Cji\right) mem}{S(i)MemCap} + \sum_{j=1}^{M} \frac{\left(Cji\right) bw}{S(i)BwCap}$$

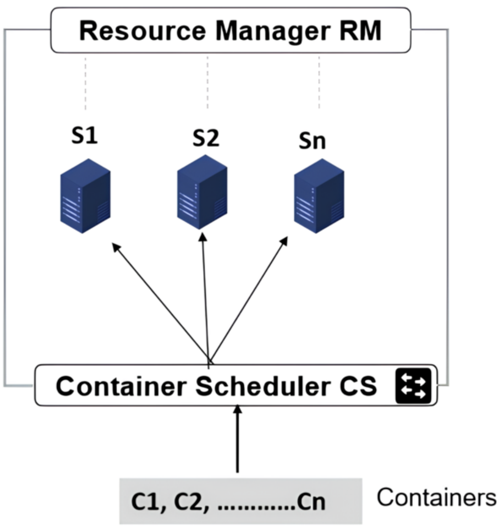
- **Container scheduler (CS):** This is responsible for analysis RU inputs, sorting data, and allocating the containers to servers according to RALB strategy. Both RALB components are presented in Figure 3.

  Algorithms 1 and 2 contain the pseudo code for implementation of RALB components.

Algorithm 1: Resource Usage (RU) Calculation Algorithm

```
Inputs:
Server_Set={Si|i = 1..N}
S_PARAMS = $S(i)_{CpuCap}$, $S(i)_{MemCap}$, $S(i)_{BwCap}$
Container_Set ={Cj|j = 1:M} containers running on servers
C_PARAMS = $C(ji)_{cpu}$, $C(ji)_{mem}$, $C(ji)_{bw}$
Output:
List of sets (Server Si, Resource Usage RU(i))
Begin
Initialize $S_{Cpu}$, $S_{Mem}$, $S_{Bw}$ to zero for all servers.
Initialize a blank list for output sets
Initialize RU, i ¬ 0
for i = 0; i < N; i++; do
    for Cji {container allocated to Server(i)} do
        $S(i)_{Cpu}$ ¬ $S(i)_{Cpu}$ +$C(ji)_{cpu}$
        $S(i)_{Mem}$ ¬ $S(i)_{Mem}$ +$C(ji)_{mem}$
        $S(i)_{Bw}$ ¬ $S(i)_{Bw}$ +$C(ji)_{bw}$
End for
$RU(i) \leftarrow \frac{S(i)Cpu}{S(i)CpuCap} + \frac{S(i)mem}{S(i)MemCap} + \frac{S(i)Bw}{S(i)BwCap}$
Insert (S(i), RU(i)) in the output list of resource usage
End for
```

Figure 3. RALB components



Algorithm 2: RALB Load Balancing Algorithm

```
Inputs:
Server_Set={Si|i = 1..N}
S_PARAMS = $S(i)_{CpuCap}$, $S(i)_{MemCap}$, $S(i)_{BwCap}$, $S(i)_{cpu\_usage}$, $S(i)_{mem\_usage}$, $S(i)_{bw\_usage}$
Container_Set = {Cj|j = 1:M} containers to be allocated
C_PARAMS = $C(ji)_{cpu}$, $C(ji)_{mem}$, $C(ji)_{bw}$
Output:
RALB Cj to Si mapping
Begin:
Call resource usage RU for all servers
Sort the list of sets (Si, RU(i)) regarding RU in ascendant way
Initialize i ¬ 0
for container Cj {waiting in the queue for allocation} do
    for i = 0; i < N; i++; do
        if $S(i)_{CpuCap}$ +$C(j)_{cpu}$ £ $S(i)_{CpuCap}$
         and $S(i)_{MemCap}$ +$C(j)_{mem}$ £ $S(i)_{MemCap}$
         and $S(i)_{BwCap}$ +$C(j)_{bw}$ £ $S(i)_{BwCap}$ then
             Allocate Cj to Si (Cji)
             $S(i)_{cpu\_usage}$ = $S(i)_{cpu\_usage}$ +$C(j)_{cpu}$
             $S(i)_{mem\_usage}$ = $S(i)_{mem\_usage}$ +$C(j)_{mem}$
             $S(i)_{bw\_usage}$ = $S(i)_{bw\_usage}$ +$C(j)_{bw}$
       else Continue
       end if
    end for
Wait 1000 ms
Repeat second loop of server local search
end for
```

EXPERIMENTAL ANALYSIS AND RESULTS

We performed a simulation of the proposed RALB algorithm in comparison to the randomized one. The required input parameters for the experiment have been set to proceed for the analysis as follows: a set of 5 servers being considered for the simulation, with their initial runtime resource usage $S(i)_{Cpu\_usage}$, $S(i)_{mem\_usage}$, $S(i)_{bw\_usage}$ as detailed in Table 1.

CPU capacity (cpu_cap), memory capacity (mem_cap), and bandwidth capacity (bw_cap) are considered to be 100 for simulation. Additionally, the pool of allocated containers for balancing purposes are being considered for each resource usage (cpu, memory, and bandwidth) use case. The

Table 1. Experiment servers initial runtime resource usage

Server	CPU_usage	Mem_usage	Bw_usage
Server 1	5	4	8
Server 2	10	4	27
Server 3	7	8	8
Server 4	11	5	30
Server 5	25	8	19

runtime requirements of containers are described in Tables 2, 3 and 4, and simulation outcomes regarding both random distribution and the RALB algorithm are displayed below each use case.

The simulation results clearly demonstrate the efficiency of the RALB algorithm in balancing the load distribution among data center servers. We conducted the simulations using a diverse set of containers (labeled A-R), each with distinct resource requirements and awaiting to be allocated most evenly by RALB. The resource aware load balancing algorithm calls the resource usage function to dynamically assess the current load status of servers regarding CPU (1), memory (2), and bandwidth (3). Additionally, it takes into consideration the actual resource requirements of the container being allocated/migrated to select the least busy server that is best suited to handle the container's resource demands (CPU, memory, and bandwidth).

Figures 4 and 5 demonstrate that the RALB algorithm exhibits a higher level of effectiveness in achieving an even distribution of the load among the five servers. The effectiveness is captured considering how much load is added to the least CPU loaded hosts before and after compared to the random algorithm (RA). The RA fails to consider the capacity of the nodes, resulting in a bottleneck scenario for servers 2 and 4. Furthermore, examining the memory usage scenario (as depicted in Figures 6 and 7), it can be seen that our proposed RALB algorithm effectively balances traffic based on the specific memory requirements of applications. We notice the allocation/migration of containers based on their resource requirements to the least busy servers (server 1, server 2, and server 5). In contrast, the RA once again fails to address this aspect, resulting in overload situations for servers 2 and 4 due to inadequate consideration of memory requirements. Last, bandwidth requirements (as depicted in Figures 8 and 9) follow the same pattern as CPU and memory, almost no overloaded or downloaded server calling for the RALB algorithm in the analysis chart; again, much traffic has been allocated to least busy servers from a bandwidth perspective. Referring to the RM component,

Table 2. CPU load analysis use case container resource requirements

Name	CPU	Memory	Bandwidth
Container A	27	10	5
Container B	30	10	5
Container C	12	10	5
Container D	7	10	5
Container E	37	10	5
Container F	13	10	5
Container G	10	10	5
Container H	9	10	5
Container I	11	10	5
Container J	10	10	5
Container K	10	10	5
Container L	12	10	5
Container M	7	10	5
Container N	17	10	5
Container O	13	10	5
Container P	10	10	5
Container Q	9	10	5
Container R	11	10	5

Table 3. Memory load analysis use case container resource requirements

Name	CPU	Memory	Bandwidth
Container A	10	30	5
Container B	10	7	5
Container C	10	12	5
Container D	10	7	5
Container E	10	37	5
Container F	10	11	5
Container G	10	14	5
Container H	10	12	5
Container I	10	13	5
Container J	10	10	5
Container K	10	12	5
Container L	10	7	5
Container M	10	17	5
Container N	10	13	5
Container O	10	10	5
Container P	10	9	5
Container Q	10	10	5
Container R	10	11	5

nodes are aware of both their resource usage and allocated containers' resource requirements. This awareness improves two major metrics for cloud infrastructure:

- **Improved quality of service (QoS):** By evenly distributing the workload among the available servers, the RALB algorithm can improve the QoS for end-users. No overloaded server means no redundant calls and faster response time from back-end servers.
- **Better resource utilization:** The RALB algorithm helps to optimize the use of resources by distributing the workload evenly among the servers considering application resource requirements. This metric enables CSP to configure their back-end servers based on their resource capabilities and to prioritize server response based on the capabilities of the datacenter regarding applications requirements.

CONCLUSION AND FUTURE WORK

The Resource-Aware Least Busy (RALB) algorithm is a novel load balancing approach specifically designed for containerized cloud environments. Through its resource-awareness and capacity-based scheduling, RALB effectively addresses the limitations of existing algorithms such as the random algorithm and least connection. Unlike previous algorithms, the RALB takes into consideration the server's capability from a resource perspective and also the container's migration time, achieved by the sorting process of the resource usage function. The sorting step of server-resource pairs (Si, RU(Si)) in the second algorithm reduces the local time search for the less loaded server, resulting in faster migration time and workload balancing. Our research demonstrates that RALB achieves a

Table 4. Bandwidth load analysis use case container resource requirements

Name	CPU	Memory	Bandwidth
Container A	10	5	30
Container B	10	5	7
Container C	10	5	12
Container D	10	5	9
Container E	10	5	27
Container F	10	5	13
Container G	10	5	12
Container H	10	5	10
Container I	10	5	11
Container J	10	5	17
Container K	10	5	13
Container L	10	5	12
Container M	10	5	10
Container N	10	5	15
Container O	10	5	15
Container P	10	5	5
Container Q	10	5	9
Container R	10	5	11

Figure 4. Random LB CPU distribution

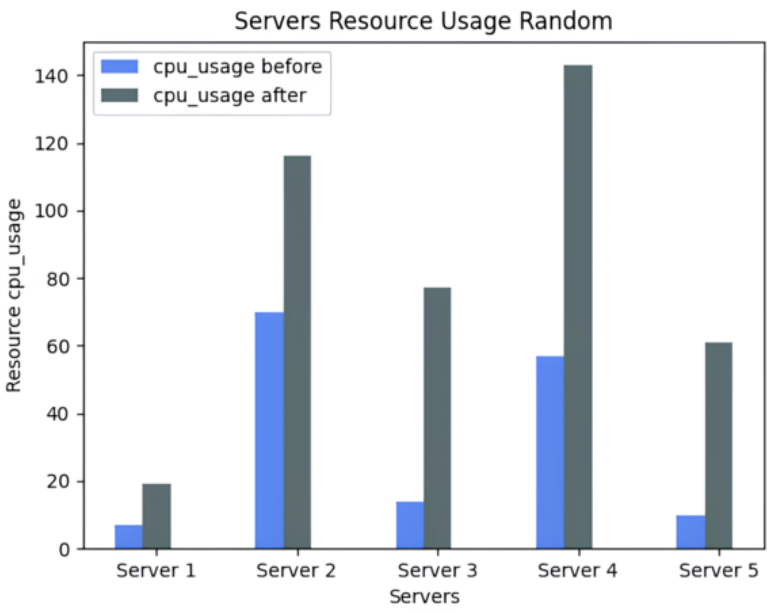


Figure 5. RALB CPU distribution

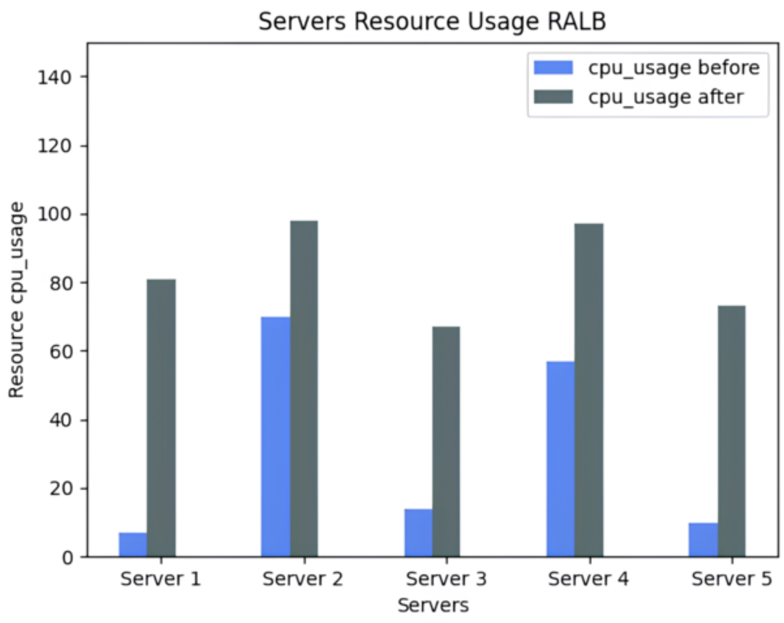
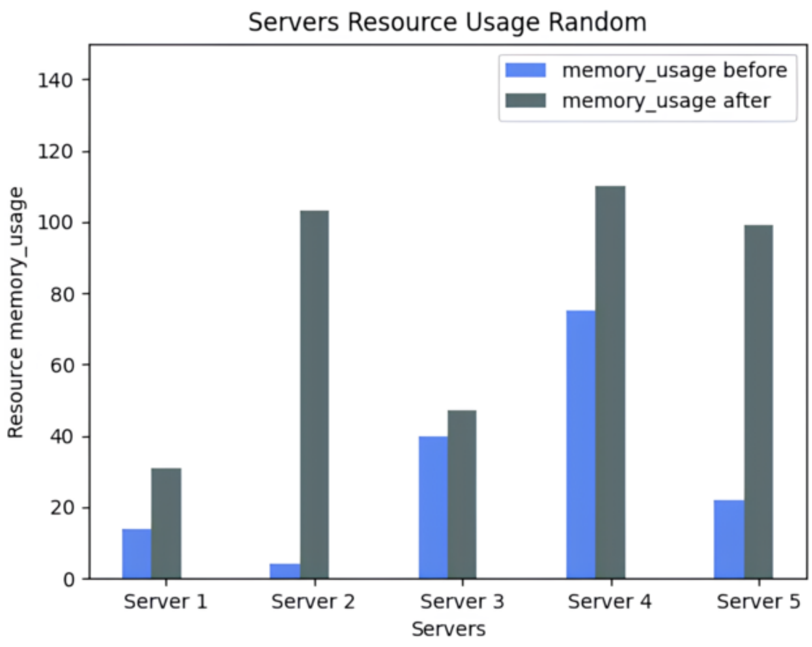


Figure 6. Random LB memory distribution



more balanced distribution of workload among servers specifically for each resource type, leading to improved performance, enhanced resource utilization, and better quality of service (QoS). The algorithm dynamically allocates and migrates containers to the least busy servers with least sufficient resource capabilities, resulting in better resource utilization and optimized response time. "Better

Figure 7. RALB memory distribution

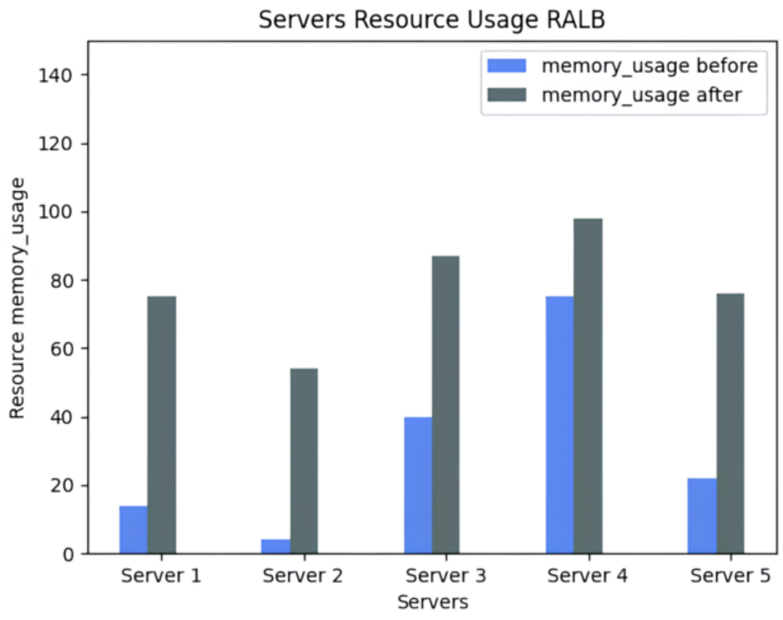
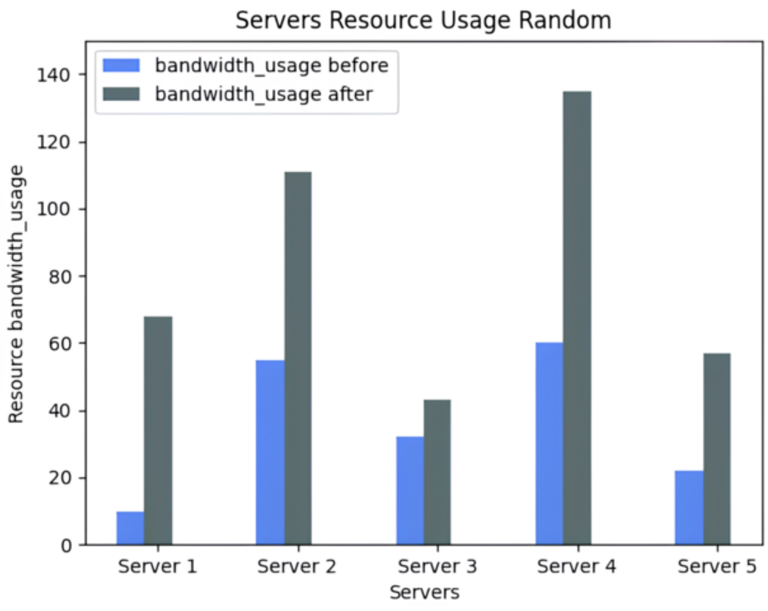
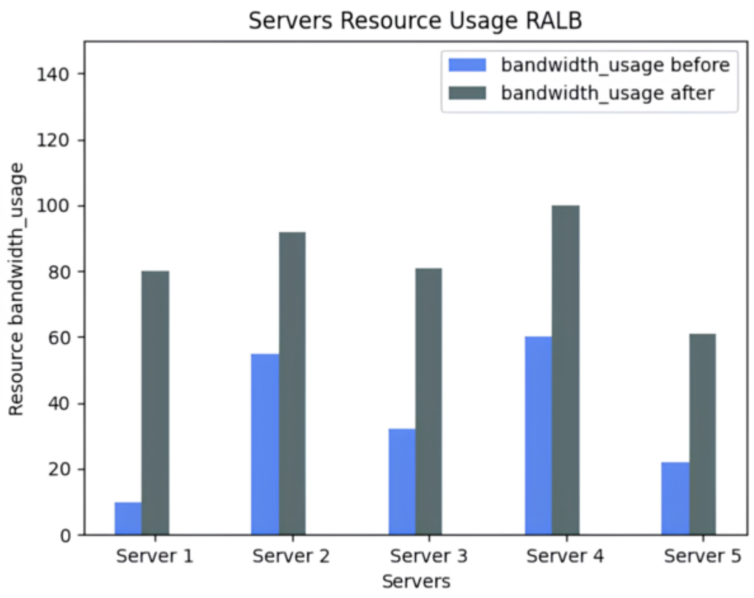


Figure 8. Random LB bandwidth distribution



resource utilization" refers to the hosting of the containers by the least sufficient server, and thus no wastage of a server's capabilities if the container is not requiring much hardware configuration, since the unutilized computing resources cost unnecessary money, reduce profit, and impact energy consumption for CSPs (Kamlesh et al., 2023). The experimental results consistently show the superiority of RALB over the random algorithm, as evidenced by the histograms and performance

Figure 9. RALB bandwidth distribution



metrics. RALB's ability to adapt to changing workloads and server capacities makes it a promising solution for modern containerized cloud environments. Further improvements of the algorithm may focus on dynamic adjustment for server's capabilities, since HW resource capacities of backend servers are considered statically configured by the cloud administrator, which impacts RALB's effectiveness in the case of a server's failure. This approach would optimize the overall fault tolerance (FT) of the containerized cloud systems adopting the proposed RALB algorithm, considering FT as one of the key elements of service level objectives (SLOs) for cloud services (Kumari & Kaur, 2023).

COMPETING INTERESTS

The authors of this publication declare there are no competing interests.

FUNDING

This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors. Funding for this research was covered by the authors of the article.

REFERENCES

Aliyu, M., Murali, M., Gital, A. Y., & Boukari, S. (2020). Efficient metaheuristic population based and deterministic algorithm for resource provisioning using ant colony optimization and spanning tree. *International Journal of Cloud Applications and Computing*, *10*(2), 1–21. doi:10.4018/IJCAC.2020040101

Guo, Y., & Yao, W. (2018). A container scheduling strategy based on neighborhood division in micro service. *2018 IEEE/IFIP Network Operations and Management Symposium* (pp. 1–6). IEEE. doi:10.1109/NOMS.2018.8406285

Kaur, K., Garg, S., Kaddoum, G., Gagnon, F., & Jayakody, D. N. K. (2019). *EnLoB: Energy and load balancing-driven container placement strategy for data centers. In 2019 IEEE Globecom Workshops (GC Wkshps)*. IEEE. doi:10.1109/GCWkshps45667.2019.9024592

Kumari, P., & Kaur, P. (2023). An adaptable approach to fault tolerance in cloud computing. *International Journal of Cloud Applications and Computing*, *13*(1), 83–106. doi:10.4018/IJCAC.319032

Lakhwani, K., Sharma, G., Sandhu, R., Nagwani, N. K., Bhargava, S., Arya, V., & Almomani, A. (2023). Adaptive and convex optimization-inspired workflow scheduling for cloud environment. *International Journal of Cloud Applications and Computing*, *13*(1), 160–185. doi:10.4018/IJCAC.324809

Lin, M., Xi, J., Bai, W., & Wu, J. (2019). Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. *IEEE Access : Practical Innovations, Open Solutions*, *7*, 83088–83100. doi:10.1109/ACCESS.2019.2924414

Manikandan, N., & Pavin, A. (2019). Comprehensive solution of scheduling and balancing load in cloud—A review. In *2019 Third International conference on IoT in Social, Mobile, Analytics and Cloud (I-SMAC)* (pp. 791–798). IEEE. doi:10.1109/I-SMAC47947.2019.9032712

Nguyen, N. D., & Kim, T. (2020). Toward highly scalable load balancing in Kubernetes clusters. *IEEE Communications Magazine*, *58*(7), 78–83. doi:10.1109/MCOM.001.1900660

Patel, D., Patra, M. K., & Sahoo, B. (2020). GWO based task allocation for load balancing in containerized cloud. In *2020 International Conference on Inventive Computation Technologies (ICICT)* (pp. 655–659). IEEE. doi:10.1109/ICICT48043.2020.9112525

Patra, M. K., Patel, D., Sahoo, B., & Turuk, A. K. (2020). A randomized algorithm for load balancing in containerized cloud. In *2020 10th International Conference on Cloud Computing, Data Science & Engineering (Confluence)* (pp. 410–414). IEEE. doi:10.1109/Confluence47617.2020.905814

Sahana, S., Mukherjee, T., & Sarddar, D. (2020). A conceptual framework towards implementing a cloud-based dynamic load balancer using a weighted round-robin algorithm. *International Journal of Cloud Applications and Computing*, *10*(2), 22–35. doi:10.4018/IJCAC.2020040102

Sureshkumar, M., & Rajesh, P. (2017). Optimizing the docker container usage based on load scheduling. In *2017 2nd International Conference on Computing and Communications Technologies (ICCCT)* (pp. 65–68). IEEE. doi:10.1109/ICCCT2.2017.7972269

Tao, X., Esposito, F., Sacco, A., & Marchetto, G. (2019). A policy-based architecture for container migration. In *2019 IEEE Conference on Network Softwarization (NetSoft)* (pp. 198–202). IEEE. doi:10.1109/NETSOFT.2019.8806659

Thongthavorn, W., & Rattanatamrong, P. (2019). Multi-container application migration with load balanced and adaptive parallel TCP. In *2019 International Conference on High Performance Computing & Simulation (HPCS)* (pp. 55–62). IEEE. doi:10.1109/HPCS48598.2019.9188218

Xu, B., Wu, S., Xiao, J., Jin, H., Zhang, Y., Shi, G., Lin, T., Rao, J., Yi, L., & Jiang, J. (2020). Sledge: Towards efficient live migration of docker containers. In *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)* (pp. 321–328). IEEE. doi:10.1109/CLOUD49709.2020.00052

Zhao, D., Mohamed, M., & Ludwig, H. (2017). Locality-aware scheduling for containers in cloud computing. *IEEE Transactions on Cloud Computing*, *8*(2), 635–646. doi:10.1109/TCC.2018.2794344