Improvement of Container Scheduling for Docker using Ant Colony Optimization

Chanwit Kaewkasi
School of Computer Engineering
Suranaree University of Technology
111 University Ave., Nakhon Ratchasima 30000
Email: chanwit@sut.ac.th

Kornrathak Chuenmuneewong
School of Computer Engineering
Suranaree University of Technology
111 University Ave., Nakhon Ratchasima 30000
Email: b5612987@g.sut.ac.th

***Abstract*—Docker, a software container implementation, has emerged not only as an operating-system level virtualization but also an application delivery platform for today Internet. However, the scheduling algorithm shipped with SwarmKit, the orchestration engine behind Docker, is suboptimal when resources are non-uniform. The use of meta-heuristic, like Ant Colony Optimization (ACO), is feasible to improve the scheduler's optimality. This paper presents a study of ACO to implement a new scheduler for Docker. The main contribution of this paper is an ACO-based algorithm, which distributes application containers over Docker hosts. It is to balance the resource usages and finally leads to the better performance of applications. The experimental results showed that workloads placed by ACO performed better than those of the greedy algorithm by approximately 15% on the same host configuration.**

I. Introduction

Software containers become the new pillars for software deployment and today Internet. Docker, a software container implementation, has emerged not only as a new virtualization technology but also an application delivery platform. Companies have reported that Docker and the software container technology are main parts of their cloud strategy [1].

Naturally, cloud vendors need scheduling algorithms to help them put containers or virtual machines into their resources as pack as possible. This kind of schedulers basically maximizes resource utilization. But software container users are different. Contributed to the stand-alone Docker Swarm project, years of its development tell us that Docker users prefer to distribute containers among their Docker hosts simply because they want their applications to yield the highest performance. Therefore the default scheduling algorithm of Docker Swarm has been changed from *binpacking* to *spread*, the reverse implementation of the binpacking algorithm [2].

Recently, Docker has introduced SwarmKit, the engine behind the new Docker's Swarm mode [3]. The scheduler in SwarmKit has been implemented using experiences learned from the stand-alone version of Docker Swarm. This scheduler is currently a kind of the round-robin algorithm, herein *greedy*, which tries to spread number of containers equally to all Docker hosts. This algorithm works fine if hosts are provisioned from the same specification, i.e. uniform. But we have found that it is suboptimal when host resources are non-uniform. The use of meta-heuristic algorithms, like Ant Colony Optimization (ACO) [4], is feasible to improve the scheduler's optimality.

ACO is a meta-heuristic algorithm which is widely adopted for optimization and combinatory problems [5], [6]. There are many works using ACO to solve problems for scheduling virtual machines. Also many works have done to solve the load balancing problem for schedulers [7], [8], [9], [10], [11]. But there is no study of using ACO to implement schedulers for a software container system yet, especially from the perspective of application workload deployment.

This paper presents a study of ACO and its practical use to implement a new container scheduler for SwarmKit and therefore Docker. The work described in this paper tries to answer the following research questions. First, is it feasible to practically use ACO as a scheduling algorithm for the container orchestration system like Docker? Second, could the ACO scheduling algorithm improve the overall performance of Docker applications?

The main contribution of this paper is an ACO-based algorithm which spread application containers over Docker hosts to better balance the overall resource usages and therefore lead to the better performance of applications compared to the current *greedy* scheduler. The experimental results found that the ACO-based algorithm performed better than the greedy one. It improved the overall application performance by approximately 15% on the same host configuration. Also from the software engineering point-of-view, we have found that the integration of ACO into SwarmKit can be done seamlessly because the semantics of task and resource reservation has been implemented in Docker 1.13-dev.

The remaining of this paper has been organized as follows. Section II reviews container and virtual machine technologies, their scheduling architectures, and reviews ACO for these resource load-balancing problems. Section III then discusses the architecture of SwarmKit, the engine behind Docker orchestration, and its modification to embed an ACO algorithm there. Section IV describes how the Docker scheduling problem was constructed, and illustrates all ACO-related equations used in our scheduling algorithm. Section V discusses the experiments and results, and finally this paper concludes in Section VI.

II. Related works

A. Containers and Virtual Machines

Container can be considered the virtualization at the operating system level. Unlike virtual machines, containers are

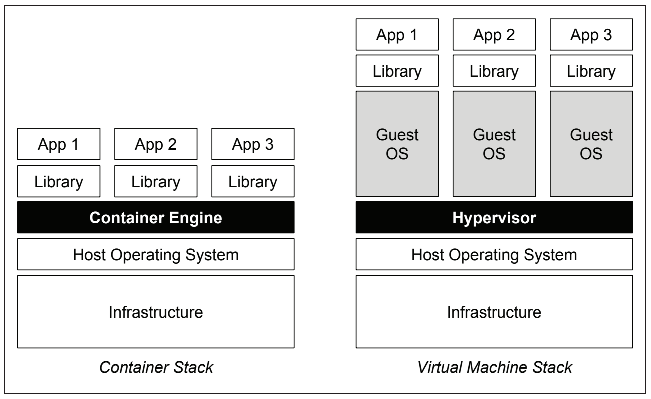

Fig. 1. Container v.s. Virtual Machine infrastructure stacks.

put into a kind of isolation which shares the host operating system's kernel. With this technique, a container's performance equals or be better than a virtual machine in almost all cases [12]. This benefit of the isolation with performance makes containers fit well for modern application deployment models. Docker is an implementation of the container management system, so called *Container Engine*. With more than 1 billion image pulls in 2015, it takes the largest market share of container engines as Docker makes containers commodity and easy to use.

Fig. 1 illustrates the comparison of a container stack versus a virtual machine stack. With the concept of container, a *Container Engine* is the resource management layer of the whole stack. Container Engine sits directly on top of the *Host Operating System*. In the world of virtual machine, this equivalent layer is *Hypervisor*. One of the interesting property of the container-centric world is that it does not require any *Guest Operating System* to run containers. Therefore the performance of containers will naturally equals or be better than virtual machines. However, in the practical cloud environments, we often mix these two stacks together as we put containers on top of a provided virtualized environment. The important here is that containers are performed well enough to run on virtual machines. This performance-wise property makes containers the application delivery platform instead of being only an operating-system level virtualization.

From the perspective of application deployment, developers who build and deploy application containers need to utilize the performance of the infrastructure as much as possible. This opposition to the cloud providers motivates the need of different kinds of container scheduling algorithms.

B. Container Scheduling

In recent years, many container orchestration systems have emerged. An orchestration system is basically a container clustering system which allows one to manage a cluster of container hosts. The orchestration always include a kind of container schedulers. For example, the stand-alone Docker Swarm [2] and Google Kubernetes [13] ships with its own monolithic scheduler. In addition to these two systems, Apache Mesos implements two-level schedulers [14].

Scheduling algorithms for clusters may have different purposes. For example, some algorithms have been designed to utilize the cluster's resources efficiently. While others, have been designed to maximize the application's performance. Google discussed three main scheduler architectures in [15]. First, the monolithic algorithm is a scheduler with the single system to handle all task placement requests. The stand-alone Docker Swarm [2] as well as SwarmKit [16] implements monolithic schedulers. The implementation proposed in this paper is also a monolithic. It was replaced the current one in SwarmKit by ACO.

The second architecture is the two-level scheduler. In this architecture, it allows the cluster to be divided into many sub-clusters. Each sub-cluster may have its own scheduler. The central resource manager acts as the second level scheduler to offer a set of resources to each individual scheduler. Apache Mesos implements this architecture [14].

The third one is the share-state scheduling architecture. A share-state scheduler is allowed to access to the cluster and let the scheduler to compete together in the form of free-for-all. It is fully distributed scheduler and there is no centralized resource allocator nor the central policy enforcement mechanism. Each scheduler are taking decision in the same way the two-level scheduling algorithm does. Google Omega is this kind of scheduler [15].

C. ACO and Scheduling Problems

There are several works adopted ACO for load balancing or task scheduling for cloud. The algorithm used in the work described in this paper is the variant of ACO, the Ant Colony System (ACS) algorithm [4].

In 2006, Zhang et al proposed the use of an ACO technique to optimize job shop scheduling problems (JSP). After applied ACS to a JSP, they concluded that the ACS was an effective method to solve JSP and the optimization could find good solutions to the problem [17].

Goyal and Singh used ACO to implement an adaptive load balancing algorithm for a grid computing environment. They compared their ACO-based algorithm with another with the simulation using GridSim. The simulation results showed that their ACO-based load balancing algorithm yielded the resource utilization better than another algorithm with standard derivation between 0.71-0.77. This work focused on the resource utilization and done only on simulation, while the work proposed in this paper focuses on a practical implementation [8].

In 2015, Tawfeek et al implemented a task scheduling algorithm for cloud based on ACO. They compared it to a First-Come-First-Serve (FCFS) and a Round Robin (RR) algorithm. From their simulation results using CloudSim, the ACO algorithm outperformed FCFS and RR algorithms in both degree of imbalance of the cluster and the average make-span of tasks over the cluster [11]. Their work focused on reducing make-span and imbalance of the resource, which is for better resource utilization of the cluster. This is contrast to our work in the way that our algorithm focuses on distributing container workloads among the container cluster to maximize the application's performance. In addition, this work done on the simulation only, while our work practically implemented the ACO algorithm for a real container orchestration system.

Gao et al developed a multi-objective algorithm using ACS to efficiently place virtual machines for the cloud computing environment. They found that their work efficiently used few minutes to solve the placement problem of a large set of virtual machines. They also successfully adopted the algorithm for multi-objective functions [7].

III. SwarmKit-ACO Architecture

A Docker cluster consists of 2 sets of nodes, managers and workers. Both are basically Docker hosts, while a manager is also responsible for management tasks including the scheduling decision. SwarmKit is actually the real engine behind the Docker cluster management. This section reviews the original SwarmKit's architecture and describes where to add the ACO algorithm to its scheduling pipeline.

Figure 2 illustrates the overall architecture of SwarmKit with ACO. A set of all worker nodes denoted *Node Set* will be processed by the *Pipeline Process Function*. These nodes will be processed through a set of filters, for example, *ConstraintFilter* will let only nodes that meet the specified constraints pass through. After that, the scheduling algorithm, *ACO* in this case (*greedy* for the original version) takes the task group and put them to the filtered nodes.

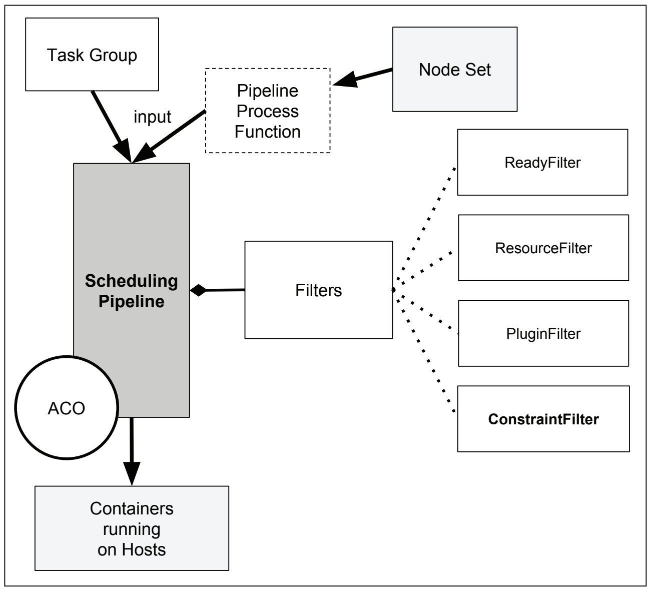

Fig. 2. SwarmKit's scheduling pipeline with the modification to add ACO.

In the recent versions of Docker, the notions of *service* and *task* have been introduced. A service consists of one or more tasks. A cluster user will start a service by submitting a number of tasks to the scheduler. The scheduler then chooses a set of nodes that fit the specified constraints and schedules the tasks to these nodes. It will always schedule a batch of tasks at a time, but not necessarily all tasks at the same time. With each batch of tasks, it is suitable to use their information to find the best plan with an ACO technique.

IV. ACO for Container Scheduling

The scheduler tries to put tasks onto the available resources. So it always reduce the available resources every time it takes an action. An artificial ant randomly looks for resources by looking at the pheromone trail from each resource. Each resource of a certain node is computed using equation 1,

$$R(j) = \psi_m \cdot \frac{r'_{m,j}}{r_{m,j}} + \psi_p \cdot \frac{r'_{p,j}}{r_{p,j}}, \tag{1}$$

where,

$R(j)$	is the resource of node j,
$r'_{m,j}$	is the available memory of node j,
$r_{m,j}$	is the total memory of node j,
$r'_{p,j}$	is the available CPU of node j,
$r_{p,j}$	is the available CPU of node j,
ψ_m	is the memory weight,
ψ_p	is the CPU weight, and

$\sum_{i=1}^{n} \psi_i = 1.0.$

To initialize the pheromone trail for each node, we use $R(j)$ in the greedy algorithm, which simply puts each task on a node in the round-robin fashion. Basically, we have,

$\tau_{0,j} = R(j)$

as the starting pheromone of each node.

Then, each probabilistic transition is computed using equation 2,

$$p(t,j) = \frac{[\tau_{t,j}]^{\alpha} \cdot [\eta_j]^{\beta}}{\sum_{i=1}^{n} [\tau_{t,i}]^{\alpha} \cdot [\eta_i]^{\beta}} \tag{2}$$

where η_j is a heuristic value of node j.

After that, the probabilistic transition $p(t,j)$ for each node is computed. The next node j is chosen using equation 3,

$$j = \begin{cases} \arg\max_{i \in N} p(t,i), & \text{if } q \geq q_0 \\ p(t,i), & \text{otherwise} \end{cases} \tag{3}$$

where $j \in P(k)$, the placement plan produced by ant k^{th}, and q_0 is the exploration rate.

The next step is for pheromone evaporation. The pheromone trail of each node will be vaporized using equation 4,

$$\tau(t) = \begin{cases} (1-\rho) \cdot \big[R(j) + \Delta\big], & \text{if } j \in P(k) \\ (1-\rho) \cdot tau(t-1), & \text{otherwise} \end{cases} \tag{4}$$

where Δ is a change of pheromone, when a task was placed to a specific resource by the scheduler. The value of Δ is always < 0. Pheromone τ is computed using equation 4 because we want to reduce the pheromone level of the chosen node significantly and make it less important for the next ant. Hence remaining containers will be placed distributedly across the resources.

Finally, the best plan, P_ω, is chosen from the most frequent plan generated by all ants using equation 5.

TABLE I. CONFIGURATION OF DOCKER HOSTS

ID	Memory	CPU
node1	512 MB	1 vCore
node2	4 GB	2 vCore
node3	1 GB	1 vCore
node4	2 GB	2 vCore
node5	512 MB	1 vCore

$$P_\omega = \arg\max_x \sum_{k=1}^{n} \big[P(k) = x\big] \tag{5}$$

Before implementing the real scheduler, we ran it as a simulation by mocking the input data and performing unit tests. We have found that the best plan has its frequency ranging from 0.35 to 0.70.

V. EXPERIMENTS

We implemented an ACO-based algorithm by modifying SwarmKit version *4dfc88c*, which is currently vendored into the Docker 1.13 master branch (1.13-dev). Both SwarmKit and Docker are written using the Go programming language. We used Go 1.6.2 as our compiler. The work was done for the x86-64 architecture as it is the major platform supported by Docker.

The experiments were conducted by forming a SwarmKit cluster on DigitalOcean, a cloud provider. The cluster consists of 1 manager and 5 workers running on the same data center. Configuration of all 5 workers are shown in Table I.

The SwarmKit manager has been set its availability from *Active* to *Drain* to prevent scheduling containers onto itself. Each worker node connects to each Docker Engine on the same host. The first experiment was done by using the original *greedy* implementation shipped with SwarmKit while another experiment was done by using the modified version of SwarmKit container our ACO scheduler. The results from these experiments denoted by `greedy` and `aco` respectively.

A. Resource Reservation

The workload is chosen to be a plain NGINX server with CPU and memory reservation as 0.5 core and 128 MB respectively. We deployed 12 NGINX tasks over the 5 nodes. Deploying uniform tasks made it is easy to observe the results of scheduling.

Fig. 3 and Fig. 4 shows the scheduling plan computed by the *ACO* and *greedy* algorithm, respectively. In both graphs, reservation resources were normalized to the biggest node, 4 GB. Task bars were divided into blocks to make it clearly how many tasks were placed in each node. Resources on the Y-axis were obtained from equation 1.

In Fig. 3, only a task was placed into each of the 512 MB node, *node1* and *node5*. *Node2* with 4 GB specification, and *node4* with 2 GB specification were used for scheduling 4 tasks each. Actually the sequence of placing in the chosen plan, P_ω, was putting a task into *node2* first then switch back and forth between *node4*, *node2* and *node3*. As also shown in Fig. 3, *node3* was used for scheduling 2 tasks.

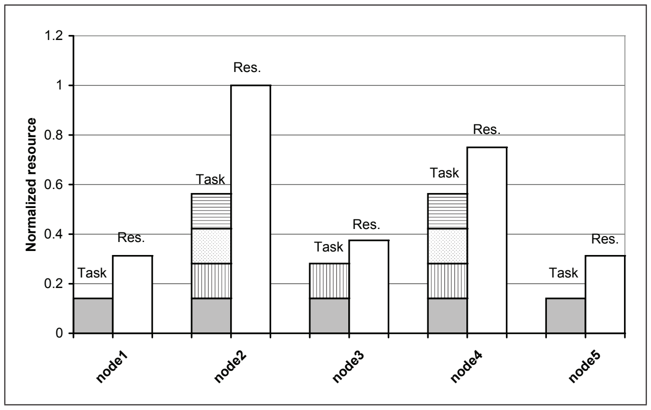

Fig. 3. Resource reservation of tasks placed by the ACO algorithm.

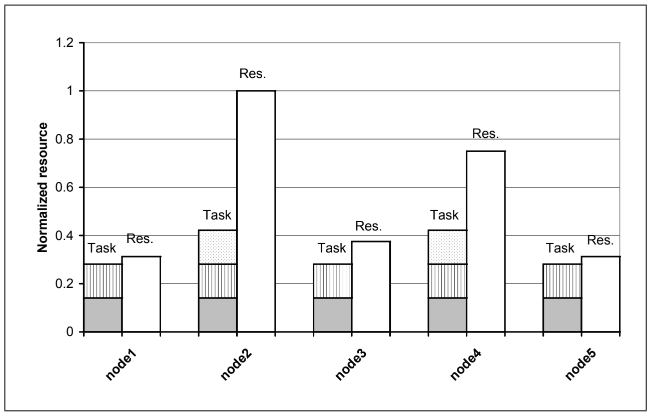

Fig. 4. Resource reservation of tasks placed by the SwarmKit's greedy algorithm.

For the SwarmKit's greedy algorithm, task distribution are illustrated by Fig. 4. The greedy algorithm distributed the first 10 tasks equally to all 5 nodes. For the last 2 tasks, it chose to put each of them to *node2* and *node4*, respectively. This resulted in *node2* and *node4* having 3 containers running on each of them.

B. Workload Performance

The results of task distribution are not the main concern. The ultimate goal we would like to obtain but cannot measure directly by looking at the task distribution is the overall performance of our application. Therefore after deployment, we measured performance of the NGINX application using the Apache Bench. The configuration use for benchmarking the application was `"-c 10 -n 1000"`.

Fig. 5 and 6 shows the overall performance of each NGINX container placed by *ACO* and the *greedy* algorithm, respectively. The presentation of these two figures must be separated because the container at each slot on both figures were *not* the same container nor running on the same host. There for they were not directly comparable. From Fig. 5, it is found that placing containers using the ACO algorithm allowed workload to be distributed better where the faster nodes served the higher requests per second. Requests per second of each container were sorted. Although, the graph cannot show which container slot ran on which host, it is obviously that, in Fig. 5, the first two containers had the lowest requests per second. This means that they were on the smallest, 512mb, hosts. Putting only one workload on an 512mb host allowed the overall performance to be better. In contrast, the greedy algorithm put two workloads on each 512mb, as clearly shown in Fig. 6. This reduced utilization of other higher performance nodes in the big picture. Table II (A) and (B) show the application workload data in requests per second and the performance percentage using the greedy algorithm as the baseline. According to the table, it is obviously that the performance of workloads placed by ACO *(A)* gained 14.80% better than the greedy algorithm *(B)*.

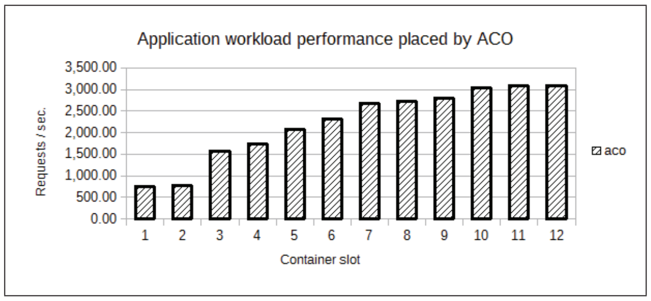

Fig. 5. Workload performance of NGINX containers placed by ACO.

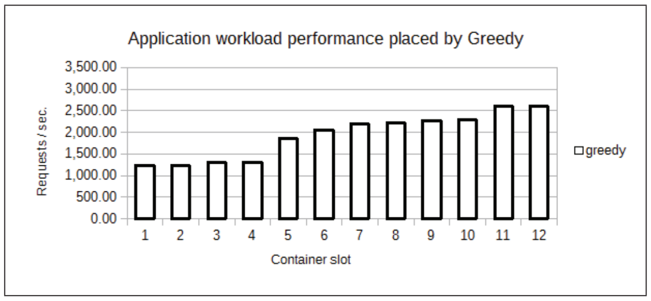

Fig. 6. Workload performance of NGINX containers placed by the greedy algorithm.

TABLE II. DATA OF THE WORKLOAD PERFORMANCE PLACED BY *(A)* ACO AND *(B)* GREEDY ALGORITHMS.

container slot	Performance of Workloads placed by ACO (req. / sec.)
1	747.28
2	770.56
3	1,569.53
4	1,726.73
5	2,071.32
6	2,304.09
7	2,674.67
8	2,728.93
9	2,791.28
10	3,043.34
11	3,083.29
12	3,085.87
Total	**26,596.89**
Percentage	**114.80**

(A)

container slot	Performance of Workloads placed by Greedy (req. / sec.)
1	1,227.45
2	1,234.87
3	1,294.46
4	1,308.66
5	1,845.66
6	2,047.27
7	2,205.86
8	2,225.50
9	2,264.46
10	2,291.75
11	2,608.54
12	2,612.92
Total	23167.41
Percentage	100

(B)

VI. Conclusion

The work described in this paper presented a use of an ACO-based algorithm for scheduling workloads in a software container environment, Docker. From the application deployment's perspective, developers would like to gain the best performance out of their application containers. To this end, they want a scheduling algorithm which helps maximizing the application performance rather than containers being fully packed. The greedy implementation of the container scheduler found in the Docker Swarm mode, which in-turn uses SwarmKit, currently distributes containers in a round-robin fashion across Docker hosts. The round-robin algorithm resulted in the sub-optimal workload performance.

This paper presented a study of ACO in the context of container scheduling. It is firstly found that the current implementation model of SwarmKit fits well with ACO. The simulation via unit testing suggested that the implementation is feasible. Therefore, the real implementation has been developed and embedded into SwarmKit version *4dfc88c* which is the current development version used by Docker 1.13-dev. According to the experimental results, the ACO algorithm proposed in this paper outperformed the original greedy algorithm. It made a real application workload, NGINX, approximately 15% faster than those placed by the greedy algorithm.

There are still a lot of rooms of improvement for the proposed ACO algorithm to perform better. Application specific parameters would be introduced to make the scheduler works better for each specific situation. Currently we used a fixed set of meta-heuristic parameters, the results of adjusting them are still yet to further study. It is also interesting to try other swarm intelligence algorithms for implementing schedulers and comparing their results in the context of application container systems. In eventually, machine learning approaches will be useful to make application containers best at utilizing their underlying resources.

Acknowledgment

This work has been sponsored by Suranaree University of Technology. The authors would like to thank the Docker Engineering team for their kindly support.

References

[1] Docker inc., "Evolution of the Modern Software Supply Chain," 2016. [Online]. Available: https://goto.docker.com/rs/929-FJL-178/images/Docker-Survey-2016.pdf

[2] V. Vieux and et al., "Swarm: a Docker-native clustering system." [Online]. Available: https://github.com/docker/swarm

[3] Docker Core Engineering, "Docker 1.12: Now with Built-in Orchestration!" Jun. 2016. [Online]. Available: http://dockr.ly/2fejvo1

[4] M. Dorigo and L. M. Gambardella, "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem," *IEEE Transactions on evolutionary computation*, vol. 1, no. 1, pp. 53–66, 1997.

[5] M. Dorigo and C. Blum, "Ant colony optimization theory: A survey," *Theoretical Computer Science*, vol. 344, no. 2, pp. 243–278, Nov. 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397505003798

[6] C. Blum, "Ant colony optimization: Introduction and recent trends," *Physics of Life Reviews*, vol. 2, no. 4, pp. 353–373, Dec. 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1571064505000333

[7] Y. Gao, H. Guan, Z. Qi, Y. Hou, and L. Liu, "A multi-objective ant colony system algorithm for virtual machine placement in cloud computing," *Journal of Computer and System Sciences*, vol. 79, no. 8, pp. 1230–1242, Dec. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000013000627

[8] S. K. Goyal and M. Singh, “Adaptive and Dynamic Load Balancing in Grid Using Ant Colony Optimization,” *ResearchGate*, vol. 4, no. 4, Aug. 2012.

[9] K. Li, G. Xu, G. Zhao, Y. Dong, and D. Wang, “Cloud Task Scheduling Based on Load Balancing Ant Colony Optimization,” in *2011 Sixth Annual Chinagrid Conference*, Aug. 2011, pp. 3–9.

[10] R. Mishra and A. Jaiswal, “Ant colony Optimization: A Solution of Load balancing in Cloud,” *International journal of Web & Semantic Technology*, vol. 3, no. 2, pp. 33–50, Apr. 2012. [Online]. Available: http://www.airccse.org/journal/ijwest/papers/3212ijwest03.pdf

[11] M. A. Tawfeek, A. El-Sisi, A. E. Keshk, and F. A. Torkey, “Cloud task scheduling based on ant colony optimization,” in *2013 8th International Conference on Computer Engineering Systems (ICCES)*, Nov. 2013, pp. 64–69.

[12] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and Linux containers,” in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Mar. 2015, pp. 171–172.

[13] D. Bernstein, “Containers and Cloud: From LXC to Docker to Kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sep. 2014.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, “Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center.” in *NSDI*, vol. 11, 2011, pp. 22–22.

[15] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” in *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013, pp. 351–364.

[16] SwarmKit contributors, “The SwamKit Project.” [Online]. Available: https://github.com/docker/swarmkit

[17] J. Zhang, X. Hu, X. Tan, J. H. Zhong, and Q. Huang, “Implementation of an Ant Colony Optimization technique for job shop scheduling problem,” *Transactions of the Institute of Measurement and Control*, vol. 28, no. 1, pp. 93–108, Mar. 2006.