IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 17, 2019, accepted March 19, 2019, date of publication March 25, 2019, date of current version April 8, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2907171

A New Infrastructure Elasticity Control Algorithm for Containerized Cloud

WALID A. HANAFY, AMR E. MOHAMED, AND SAMEH A. SALEM
Department of Electronics, Communications, and Computers, Faculty of Engineering, Helwan University, Helwan 11792, Egypt
Corresponding author: Walid A. Hanafy (walid_ashraf@h-eng.helwan.edu.eg)

ABSTRACT In the last decade, containers have become a superior alternative to hypervisor-based virtualization. Containerization has revolutionized data centers from being an infrastructure-oriented to be application oriented. Modern cloud consumption patterns such as flash crowds require a certain amount of elasticity that is realized with controlling the amount of provisioned resources autonomously. Cloud elasticity is significant as it influences the performance of utilized resources, service level commitment, and power consumption. In this paper, an infrastructure elasticity control algorithm for a containerized cloud is proposed. The proposed algorithm augments the load balancing criterion with elasticity control. Several experiments with various metrics are carried out to examine the performance of the proposed algorithm. The results demonstrate the superiority of the proposed algorithm and the effects of elasticity across various measures.

INDEX TERMS Containers, cloud computing, elasticity control, containers migration, load balancing.

I. INTRODUCTION

Container virtualization technology, which is also called lightweight operating system virtualization, have become state of the art cloud deployment model [1]. Container engines provide a shared operating system environment that increases server consolidation percentage. A Linux container mainly relies upon two kernel features namely cgroups and namespaces [2]. Cgroups [3] provides resource limitation facilities while namespaces provide resource isolation capabilities.

As shown in Figure 1, containers utilize the namespaces and cgroups kernel features and the union file system. Union file system [4] stacks multiple images to form the final view for the container. An image is a read-only file system layer that adds up to form the files seen by a container. Container images are stored in a container registry machine where all images and their hierarchy information are mapped and saved there. Containers gained instant success due to their high utilization efficiency, fast start-up time and performance supremacy [5].

Currently, organizations are racing to employ containers in various applications that include applications deployment, application middleware and cloud platforms [6]–[8]. Aside from these basic container employment techniques, containers have emerged as a best practice deployment model in fog computing and IoT platforms [9], [10]. The advancements in containers and containers platforms allowed the appearance of containerized cloud infrastructures [11] and Containers as a Service ''CaaS'' computing model [12] that acts as a middle layer between Infrastructure as a Services ''IaaS'' and Platform as a service ''PaaS.''

Modern applications and computing services are built with considerations for rapid change in the number of active customers known as flash crowds [13]. Daily deals and time-limited offers became a widely adopted strategy by companies [13]. The offers usually accompanied by an unpredicted increase in the number of active customers. This rapid increase in the number of customers requires a level elasticity that is not tackled by current provisioning mechanisms.

Al-Dhuraibi *et al.* defined elasticity as the ability to scale autonomously in an optimal manner [14], as optimality means time and provisioning efficiency. They classified different scopes, approaches, and methods for elasticity in cloud computing. According to them, authors perform elasticity control on either application/platform or infrastructure level. Infrastructure elasticity is the alteration in the provisioned physical resources. They also stated that elasticity could be classified into vertical and horizontal elasticity. Vertical elasticity, known as resizing, the application resources in terms of CPU,

The associate editor coordinating the review of this manuscript and approving it for publication was Gaurav Bhatia.

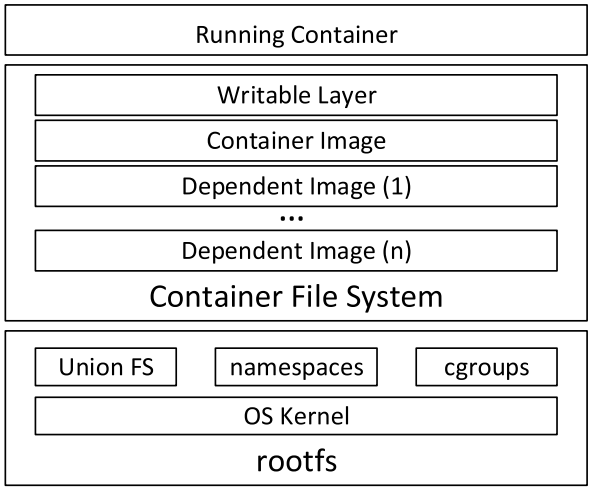

FIGURE 1. Container Structure.

Memory, and IO are altered at runtime. On the other hand, Horizontal elasticity, known as replication, is the process of adding or removing resources instances.

Kan proposed a horizontally elastic web deployment platform by utilizing proactive/reactive elasticity controllers [6]. The proactive controller exploits a prediction algorithm to anticipate the required load while the reactive acts as a safety valve for sudden load changes. Also, Al-Dhuraibi *et al.* [2], proposed an application level vertical elasticity controller. The proposed controller update the allocated resources if it surpassed its upper or lower bounds. Moreover, if the host reaches its limits, the elasticity controller executes a live container migration.

Piraghaj *et al.* introduced an infrastructure elasticity management algorithm [15]. Such algorithm utilizes a two-layer virtualization scheme, where containers run in virtual machines "VMs" rather than directly on the host itself. The algorithm tries to maximize the power efficiency by adjusting the number of utilized machines. The proposed method migrates containers from unstable to stable machines where upper and lower thresholds define stability. However, the aforementioned elasticity methods were not designed to cover flash crowds scenarios. Additionally, elasticity effects on other metrics such as load balancing, resource utilization, service level commitment, and power efficiency were not studied extensively.

Eager *et al.* introduced a load balancing algorithm that works iteratively [16]. The algorithm consists of a sender and receiver initiated (Push and Pull) load sharing methods based on queue length. As a result of its low complexity and performance superiority, this algorithm has been adopted vastly in parallel systems, distributed systems, and cloud computing. For example, Forsman *et al.* adopted the algorithm in a virtualized data center [17].

This paper proposes an infrastructure elasticity control algorithm based on eager's load balancing algorithm by altering the main procedures to adapt to flash crowds elasticity requirements. The proposed algorithm considers two agents, namely the master and host agents. The master agent is responsible for elasticity and coordination between hosts while the host agent is responsible only for its host well-being. The host agent monitors and predicts its utilization using Autoregressive Moving Average "ARMA" [18] which derives its state. In the case of a non-normal state, the host sends the appropriate request to the master that initiates an auction against other hosts. The auctions are based on multiple selection criteria and among a subset of the available hosts. The master performs elasticity by handling failures in load interchange scenarios.

The rest of the paper is structured as follows. In section II, the notations, assumptions, and problem formulation are discussed. Section III, discuss system components and hosts' states. In section IV, the proposed work and evaluation methodology are discussed. Section V includes the results and discussions. Finally, Section VI presents the conclusion and future work.

II. NOTATIONS, ASSUMPTIONS, AND PROBLEM FORMULATION

A. NOTATIONS

Assume a container set C $c_1, c_2, c_3, \ldots, c_N$ where a container c_i is based on a container image I_j where container images are located on a machine container registry machine r. Containers run on a host machine h. Let H denotes the host machines set$\{h_1, h_2, \ldots, h_N\}$, where H can be either homogenous or heterogeneous.

Given a host machine h_i state denoted by s_i, where $s_i \in \{OFF, IDLE, UNDER-UTILIZATION, NORMAL, OVER-UTILIZATION, EVACUATION\}$. $Util\,(h_i) \rightarrow R$ computes the utilization for a given host h_i. Additionally, $Cost\left(H_x, C_y\right) \rightarrow R$ receives the hosts and containers sets and returns the overall cost. Finally, all host machines are controlled by a master machine$m$.

B. ASSUMPTIONS

The network throughput is modeled according to Mathis equation [19], throughput T is calculated as,

$$T = \frac{MSS * C}{RTT \times \sqrt{p}} \tag{1}$$

where the equation utilizes the Maximum Transmission Segment "*MSS*," C a constant that equals $\sqrt{3/2}$, Network Round Trip Time "*RTT*," and packet loss probability "p." The network throughput defines the time taken by all data transfer operations such as image loading and migrations across the network.

C. PROBLEM FORMULATION – OPTIMAL ELASTICITY CONTROL PROBLEM

Given 2-tuple$< H, C >$, find the optimal migration policy at given time$T$, such that:

$$Cost^*\left(H_l^*, C\right) = \min_{H_t \subset H} \arg\ Cost(H_t, C) \tag{2}$$

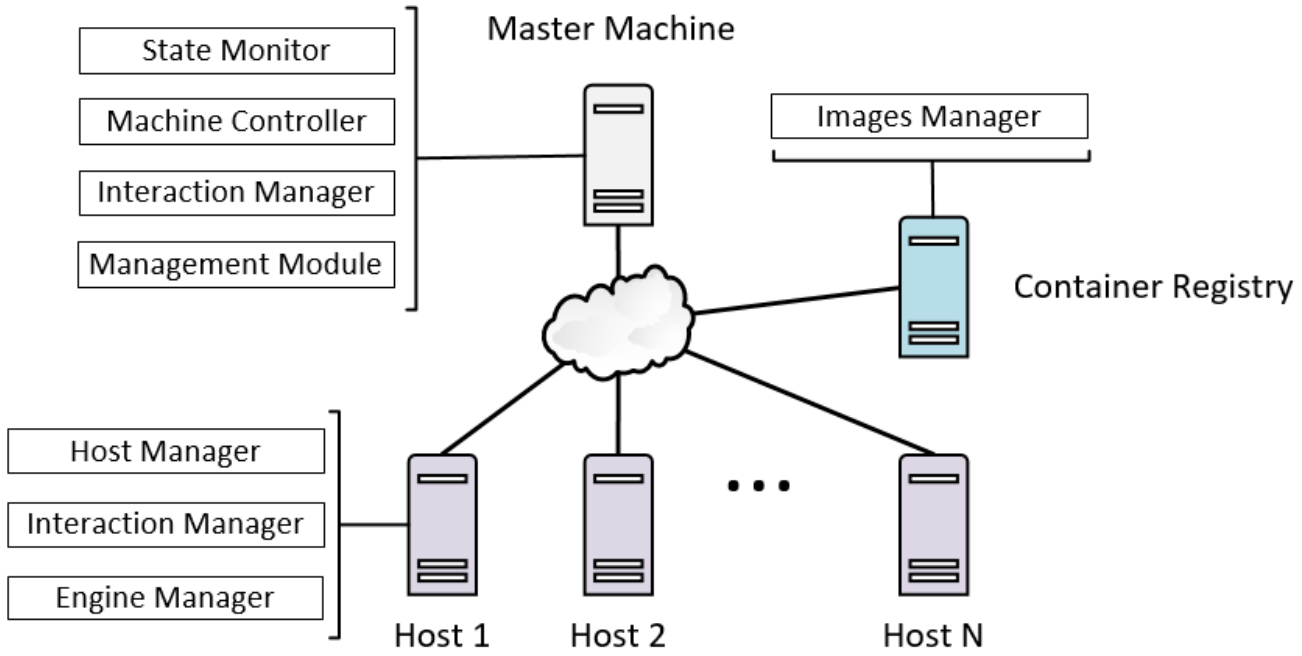

FIGURE 2. Infrastructure Components.

III. COMPONENTS AND STATES

A. SYSTEM COMPONENTS

The adopted containerized cluster is composed of the three types of machines namely, master, host and container registry. Every machine is packed with a management agent; for ease, the machine and agent keywords are used interchangeably. Each agent is composed of multiple components as shown in Figure 2. The proposed master agent has four main modules:

- The *State Monitor* module acts as a hosts' state information hub where it knows all hosts' states and configuration all the time.
- The *Machine Controller* is responsible for controlling the hosts' power state.
- Also, the *Interaction Manager*is responsible for communication and communication validity between the master machine and host machines. Besides, it delimits the number of active requests and responds automatically in case of congestion.
- Finally, The *Management Module* contains management and elasticity policies. In addition, it acts as a coordinator between different master tasks.

The host machine is the actual worker that executes and hosts containers and follows master agent commands. The host agent main modules are:

- The *Host Manager* is responsible for the wellbeing of the host itself. It triggers load absorption/eviction requests, evaluates the benefit of an incoming request and reports changes to the master agent.
- The *Interaction Manager* is responsible for communication management. For example, it rejects requests when a host is participating in an auction either as an owner or as a bidder.
- The *Engine Manager* is responsible for controlling the containerization engine.

Finally, the container registry is an image store where the image manager module is responsible for managing images, maintaining images' trees, and reporting them to hosts.

B. HOST STATES

Each host h_i, contains a set of containers C_i where the load is modeled as a collection of resources such as CPU, RAM, and Network IO. The load of a host h_i is denoted by l_i, where

$$l_i = l\left(C_i\right) + l\left(Os_i\right) \tag{3}$$

$l\left(C_i\right)$ is the total load of containers hosted at host h_i and $l\left(Os_i\right)$ is the base load occupied by the OS. However, the considered host's load is the ARMA [18] predicted load rather than the current load. The predicted load denoted by$l(t+1)$ is calculated as follows:

$$\begin{aligned} l\left(t+1\right) = \beta \times l\left(t\right) + \gamma \times l\left(t-1\right) \\ + \left(1-\left(\beta+\gamma\right)\right) \times l\left(t-2\right) \end{aligned} \tag{4}$$

given that, $\boldsymbol{\beta}$ ***and*** $\boldsymbol{\gamma}$ are the prediction coefficients. Host $\mathbf{h_i}$ utilization is denoted by $\mathbf{u_i}$,

$$UTIL\left(h_i\right) = u_i = \frac{l_i\left(t+1\right)}{\max l_i} \tag{5}$$

where utilization is the ratio between the predicted load and max load.

Figure 3, illustrates the transition of the host machines between states. The set of hosts in the $OFF, IDLE,$

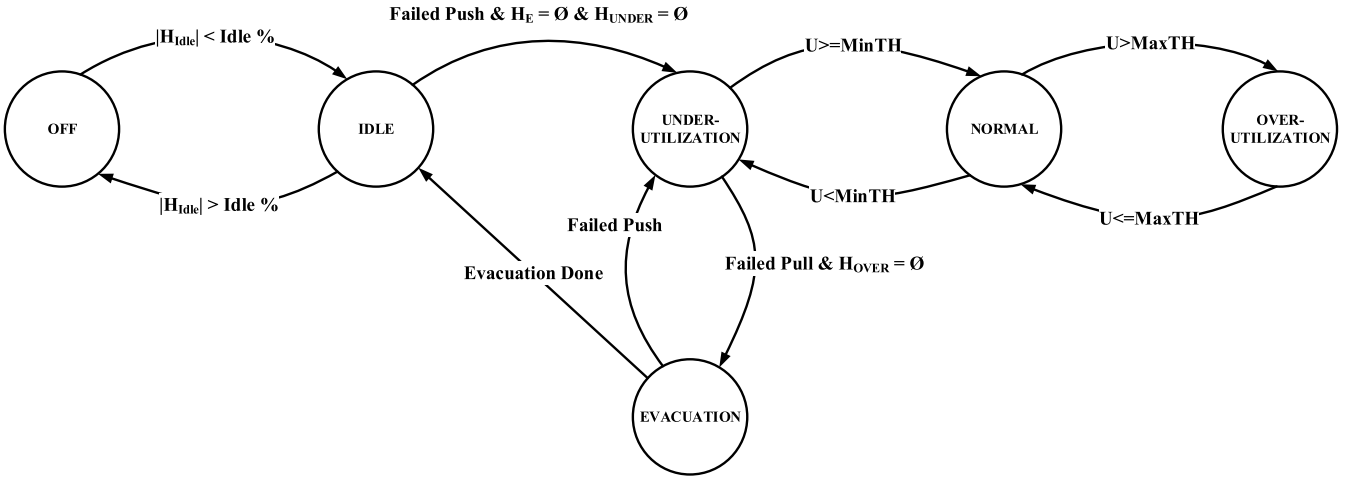

FIGURE 3. Hosts' States.

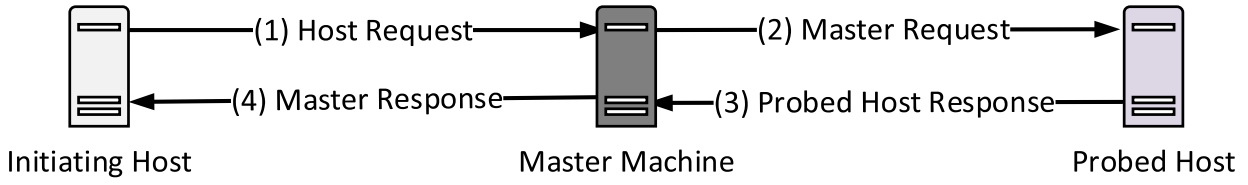

FIGURE 4. Algorithm's Events Diagram.

UNDER − UTILIZATION, *OVER − UTILIZATION*, *and EVACUATION* states are denoted by H_{OFF}, H_{IDLE}, H_{UNDER}, H_{OVER}, *and*H_E respectively, where $|H_{IDLE}|$ is maintained as a percentage of the operating machines. Moreover, the relations between the *UNDER − UTILIZATION*, *OVER − UTILIZATION* states are determined based on the minimum and maximum thresholds denoted by *MinTH* and*MaxTH*.

IV. METHODOLOGY

A. PROPOSED ALGORITHM

This work proposes an elasticity control algorithm based on Eager *et al.* load balancing algorithm [16]. Initially, the algorithm separates the utilized load sharing strategy, to be either Push or Pull. However, this paper proposes a client-server architecture that allows the utilization of both strategies simultaneously.

Figure 4 describes the algorithm's sequence of events. As shown, the host initiates the cycle by sending a request to the master according to *Algorithm* 1. The master handles the requests using *Algorithm* 2 and executes *Algorithm* 3 to perform elasticity tasks. Finally, a probed host follows *Algorithm* 4 in order to either participate or reject an auction.

Algorithm 1 describes the actions taken by a free host in a none-normal state. The host's state derives the action it takes. As shown, if the host is *UNDER − UTILIZED*, it demands the master to send it a container to correct its state (Pull Request). Conversely, an *OVER − UTILIZED* scenario where the host demands the master to redistribute some of its load (Push Request). However, only in the case of Push request, the machine selects a container using Container Selection Policy denoted as *CSP* currently set to ratio between container utilization and migration count to avoid containers exhaustion.

Algorithm 1 Container Structure

Input : $Host(\boldsymbol{h}_i)$, $HostContainers(C_i)$
Begin
  $S \leftarrow GetState(h_i)$
  if $S = UNDER - UTILIZED$ ***then***
    $SendPullRequest()$
  else if $S = OVER - UTLIZED$***or***$S = EVACUATION$ ***then***
    $C_i^x \leftarrow \max_{c_i} \arg\ CSP(C_i)$
    $SendPushRequest(C_i^x)$
  $R \leftarrow MasterResponce()$
  if $R.GetState() = Success$ ***then***
    $ExecuteMigration()$
    if $S = EVACUATION$ ***and*** $c = \emptyset$ ***then***
    UpdateState (h_i, $IDLE$)
  else
    if $(R.hasNewState())$***then***
      UpdateState (h_i, $R.GetNewState()$)
    $BackOff()$
  end

Algorithm 2 Master Handeling Algorithm

Input : *HostRequest* (r_i) , *Hosts*(S)
1: ***begin***
2: $X \leftarrow \sigma_{\mathrm{LT,TT}}(\mathrm{H})$
3: ***if*** $X = \emptyset$ ***then***
4: *Exectute* ***Algorithm 3***
5: ***else***
6: *Execute auction across X*
7: $V \leftarrow$ *Valid Bids*
8: ***if*** $V \neq \emptyset$ ***then***
9: *find* b_x *using* HSP *where* $b_x \in V$
10: *respond to* h_i *by* b_x
11: ***esle***
12: *Exectute* ***Algorithm 3***
13: ***end***

A successful response indicates that the host will push or pull a container. Nonetheless, an evacuating host that ejected all its containers informs the master machine to elasticize the infrastructure down. On the other hand, a rejection response can be accompanied by a state update to start or stop evacuation. Also, rejections lead to the execution of a binary exponential backoff algorithm similar to the one used in wireless networks [20].

As depicted in Figure 4, A host request reaches the master which reacts according to *Algorithm* 2. The master initiates the auction against a subset of the available hosts (candidates list). These hosts are filtered against Logical and Testing Thresholds denoted by *LT* and *TT* respectively. *LT* is executed according to the auction type and the host's state. So, in the case of a Push Request, the candidate list is filled with underutilized hosts; alternatively if none was found, it is filled with normal hosts. On the other hand, in case of a pull request, the candidate list is filled with overutilized hosts then with normal in case none was found. In either case, the percentage of the tested hosts among available and matching candidates is filtered by TT. $TT \in \{25\%, 50\%, 75\%, 100\%\}$. Such as, in the case of 100 host with 25 % as the TT, only 25 hosts are probed.

After performing the auction, the *Management Module* then tries to find an approving host according to Host Selection Policy denoted as *HSP*. Selection policy defines the selection criteria between valid bids only. The proposed algorithm utilizes a different set of host selection policies namely Least Full, Most Full, Least Pulls, and Random Selection. Finally, the master implements elasticity by remediation, such as if no candidates or no valid bids was found *Algorithm* 3 is put into action.

Algorithm 3 describes the actions taken by the master machine to elasticize the infrastructure. The elasticity management algorithm acts differently according to the request type and the host state. The *PowerController* elasticize the infrastructure up or down. Also, rejections can be accompanied by an update in the requested state. Moreover, in some cases, the master returns a rejection without executing any actions.

Finally, the tested (probed) hosts act based on *Algorithm*4. The host sends a valid bid only if the container addition or removal process does not change its state. The migration is initiated if the auction results in the selection of its bid.

Algorithm 3 Elasticity Managemnt

Input : *Host Request* (r_i) , *Hosts*(H)
1: ***begin***
2: $action \leftarrow NONE$
3: $t \leftarrow r_i.GetType()$
4: $s_{new} \leftarrow r_i.GetRequesterState()$
5: ***if*** $t = Push$ ***then***
6: ***if*** $(h_i \in H_E)$ ***then***
7: $action \leftarrow (Sets_{new} toUnder - Utilized)$
8: ***else if*** $H_E \cup H_{UNDER} = \emptyset$ ***then***
9: $action \leftarrow Activatehost(h_j)$
10: ***else***
11: ***if*** $H_{UNDER} = \emptyset$ ***then***
12: $action \leftarrow (Sets_{new} toEVACUATION)$
13: *Executeaction* and *Reject* r_i
14: ***end***

Algorithm 4 Probed host Procedure

Input:$Request\ (r_j)$, *Host* (h_i) , *Containers* (C_i)
1: ***begin***
2: $t \leftarrow r_i.GetType()$
3: $B \leftarrow NONE$
4: ***if*** $t = Push$ ***then***
5: $C_j^x \leftarrow r_i.GetContainer()$
6: ***if*** $Util\left(h_i + C_j^x\right) < MaxTH$ ***then***
7: $B \leftarrow ValidBid$
8: ***else***
9: $C_i^x \leftarrow \max \arg_l imitsc_i CSP(C_i)$
10: ***if*** $C_i^x \neq null$ ***and*** $Util\left(h_i - C_i^x\right) \geq MinTH$ ***then***
11: $B \leftarrow ValidBid$
12: *respond by B*
13: $R \leftarrow$ *Master Responce*
14: ***if***$(R.GetResponce() = Selected)$***then***
15: *ExecuteMigration*()
16: ***end***

B. EVALUATION METRICS

Infrastructure elasticity affects multiple data center metrics. For example, the change in provisioned resource changes the load balancing, network traffic, and utilized power. For this reason, this paper illustrates the performance of the elasticity and its repercussions. Elasticity can be evaluated by measuring the adherence between actual and required resources. We propose using Root Mean Square Error "RMSE" in measuring elasticity where RMSE determines the difference between needed and actual provisioned machines. RMSE is

denoted as follows:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=0}^{N}(H_i - \hat{H}_i)^2} \quad (6)$$

H_i is the actual number of hosts, $\hat{H}_i$ is the optimum number of hosts and N is the number of samples. Elasticity affects the number of hosts, which shapes the amount of power consumption of the data center. The power consumption is not measured per host individually but measured by the total data center consumption$P_{dc}(t)$,

$$P_{dc}(t) = \sum_{1}^{N} P_i(t) \quad (7)$$

where N is the total number of active hosts, and host power consumption P_i is calculated as follows,

$$P_i = (P_{max} - P_{idle}) \times \mu + P_{idle} \quad (8)$$

P_{max} is the maximum consumed power, P_{idle} is the idle power, and μ is the resource utilization percentage. Additionally, operations like elasticity require migration and have multiple effects on the infrastructure network. Therefore, the effect on the network traffic will be measured in multiple metrics namely, total migrations, total message, pull requests, and total transmitted data.

In general, the elasticity aims for service level agreement ''SLA'' compliance management, in this work SLA is measured using two metrics namely containers downtime and utilization SLA. The containers Downtime is the total time the container was paused during the migrations attempts. While the utilization SLA in a cloud data center was defined by [21] as an indicator of compliance between the actual and provisioned resources. SLA is the ratio between needed and actual resources and the following equation is used to calculate the SLA,

$$SLA = \sum_{i=1}^{N}\sum_{j=1}^{M_i} \frac{u(h_i, c_j)_r - u(h_i, c_j)_a}{u(h_i, c_j)_r} \quad (9)$$

$u(h_i, c_j)_r$ and $u(h_i, c_j)_a$ are required and allocated resources for container c_j on host h_i and N is the number of hosts and M_i is containers per host h_i.

Lastly, Entropy was adopted by [17] to measure the infrastructure Load balancing capabilities. Entropy measures load distribution using normalized entropy to compute a value independent of the number of machines. The entropy of a perfectly balanced load over multiple hosts approaches one. The normalized entropy $E(H)$ is computed as follows,

$$E(H) = \frac{-\sum_{i=1}^{N} u_i log u_i}{\log N} \quad (10)$$

where u_i is the utilization percentage of host h_i and N is the number of hosts. In conclusion, these metrics presents conclusive measures of the data center performance under the effect of any elasticity algorithm.

V. EXPERIMENT AND RESULT

The proposed algorithm is evaluated against the algorithms proposed by Forman *et al.* [17] and Zhao and Huang [24] across the metrics as mentioned earlier.

Forsman's method is based Eager's Algorithm, where a suffocating or an underutilized machine initiates an auction across all machines. The auction selects successful bids upon its load balancing profitability. Additionally, they utilize Exponentially Weighted Moving Average ''EWMA'' for load prediction.

On the other hand, Zhao and Huang [24] incubates a periodic compare and migrate approach. At each period, all machines check whether the existence of VMs at other machines will benefit the overall infrastructure balance. Zhao's algorithm shares some similarities with Forsman; as it: adopts a master-less philosophy, tests all available hosts, and lacks the ability to elasticize. However, only the work done by Zhao contains a shared memory model, implements all applicable migrations, and does not implement any load prediction mechanisms.

A. EXPERIMENTAL SETUP

All algorithms and policies are simulated using Container Simulator [22] that simulates native container data center where an isolated entity simulates each machine, container, and policy. The data center network follows a fat tree architecture adapted from [23]. The experiment assumes a set of Containers C distributed across hosts H with a preconfigured start utilization percent.

The algorithm tries to elasticize the infrastructure according to the preconfigured policies. The algorithm is benchmarked against Forsman and Zhao. Then the effects of the Host Selection Policies are assessed. Finally, the testing threshold influence on the algorithm performance is evaluated.

TABLE 1 shows different simulation trials. These trials cover flash crowd behaviors using load burst and drain scenarios. Each trial a combination of start utilization and mid-time action. The start utilization equals$X + U(-n, n)$, where $X$is the utilization median value and n is the displacement value which is set to 20. For the mid-time, the simulation manager subjects containers to either burst or drain. These events are implemented by subjecting half the containers to a 50% sudden change in load demand.

Finally, TABLE 2 describes the simulation parameters such as containers' and hosts' size, and the number of hosts. The parameters also include power coefficients implemented from [25] and ARMA prediction coefficients adopted from [18]. Also, the network delay model is computed using Mathis model [19] where it mainly depends on the probability of packet drop (P) and the Round Trip time (RTT), and realistic parameters are adopted from [23]. Finally, the simulation is considered a 6-hour snapshot of a Container-Service cluster where the measures are collected every one min, and the experiments are repeated for 30 times for accurate results.

TABLE 1. Simulation scenarios.

Scenario Name	Start Utilization	Mid-Time Action
Mid-Utilization with Burst effect	50%	Burst
High-Utilization with Drain effect	70%	Drain

TABLE 2. Simulation parameter settings.

Item	Value
Container CPU	N($\mu = 4$, $\sigma = 1$) BIPS
Container Memory	U(256, 512, 768, and 1024) MB
Container Image Size	N($\mu = 50$, $\sigma = 20$) MB
Number of Images	2000
Host CPU	100 BIPS
Host RAM	32 GB RAM
Number of Hosts	200
Utilization Boundaries ($MinTH,\ MaxTH$)	0.7 and 0.9
Power Coefficients ($P_{max},\ P_{idle}$)	124, 239 Watt/hour
Prediction Coefficients (β, γ)	(0.8, 0.15)
Packet Drop Rate (P)	0.01%
Round Trip Time (RTT)	400 μS
TCP Header	40 Byte
Maximum transmission Unit (MTU)	1460 Byte
Maximum Segment Size (MSS)	MTU + TCP Header = 1500 Byte
Sampling time	1 min
Simulation Time	6 hours
Number of Trials	30

TABLE 3. Effect of selection policies in scenario 1 (mid-utilization with burst effect).

Policy	Least Full	Least Pulls	Most Full	Random
Elasticity (RMSE)	25.95	27.54	28.96	26.82
Power (KWH)	51.73	52.26	52.64	52.23
Load Balancing (Average Entropy)	0.9982	0.9983	0.9981	0.9983
Containers Migrations	1373	1250	1249	1335
Containers Image Pulls	5041	2791	4587.	4897
Total Message (10^3)	279	251	382	264
Transferred Data (GB)	1138	948	1039	1105
Containers Average Downtime (sec)	3.39	3.10	3.10	3.28
SLA Violation %	0 %	0 %	0 %	0 %

B. BENCHMARKING PROPOSED ALGORITHM

The algorithm performance is evaluated against Forman and Zhao infrastructures management algorithms. Figure 5 illustrates the proposed algorithm with its peers in the first scenario (Mid-Utilization with Burst effect). Figure 5(a) shows that the proposed work achieves better elasticity and therefore better power efficiency as shown in Figure 5(b). Figure 5(c), illustrates the abilities of each algorithm to balance the infrastructure where it shows a minor delay in the proposed algorithm.

Figure 5 (d) shows the total messages accounted by each algorithm, where the proposed work and Zhao send the least and most amount of messages respectively. Figure 5 (e), shows the total number of migrations performed by each algorithm. Figure 5 (f, g, h), shows the side effects of the migrations in terms of downtime, image pulls and consequently, to the total migrated data. Moreover, Figure 5 (i) describes the SLA violation percentage that mostly equals zero due to the nature of the simulation case itself.

The proposed algorithm outperforms its peers in power metrics due to its ability to adapt the number of hosts according to the needed load. On the other hand, Forsman and Zhao do not execute elasticity policies which maximize the utilized power. Moreover, the proposed algorithm performs the least number of messages due to the logical and threshold filtering methods. Also, it accounts less entropy, more migrations, more downtime, image pulls and transferred data because it commits elasticity which accounts more migrations and therefore more network usage.

Forsman sends a large number of messages as it executes an auction against all host in every request. Moreover, Forsman's profitability based selective migrations benefits them

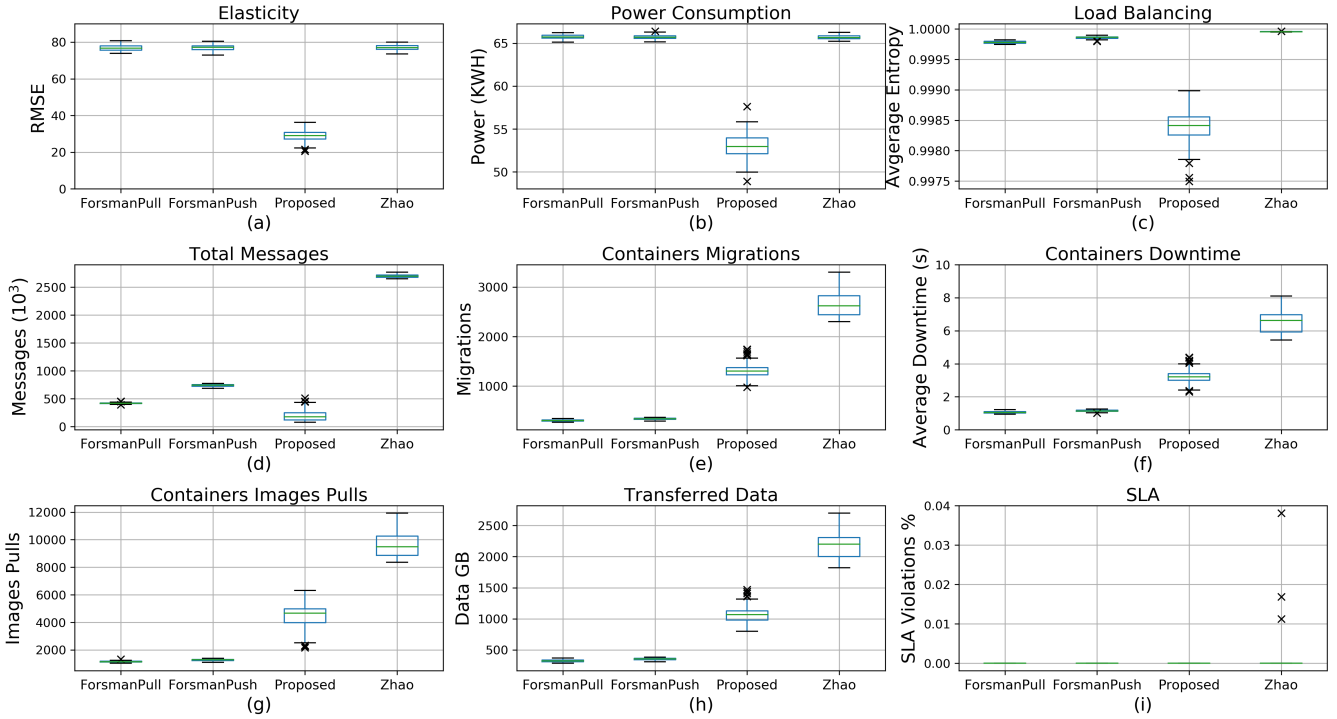

FIGURE 5. Algorithms comparisons (Mid-Utilization with Burst effect).

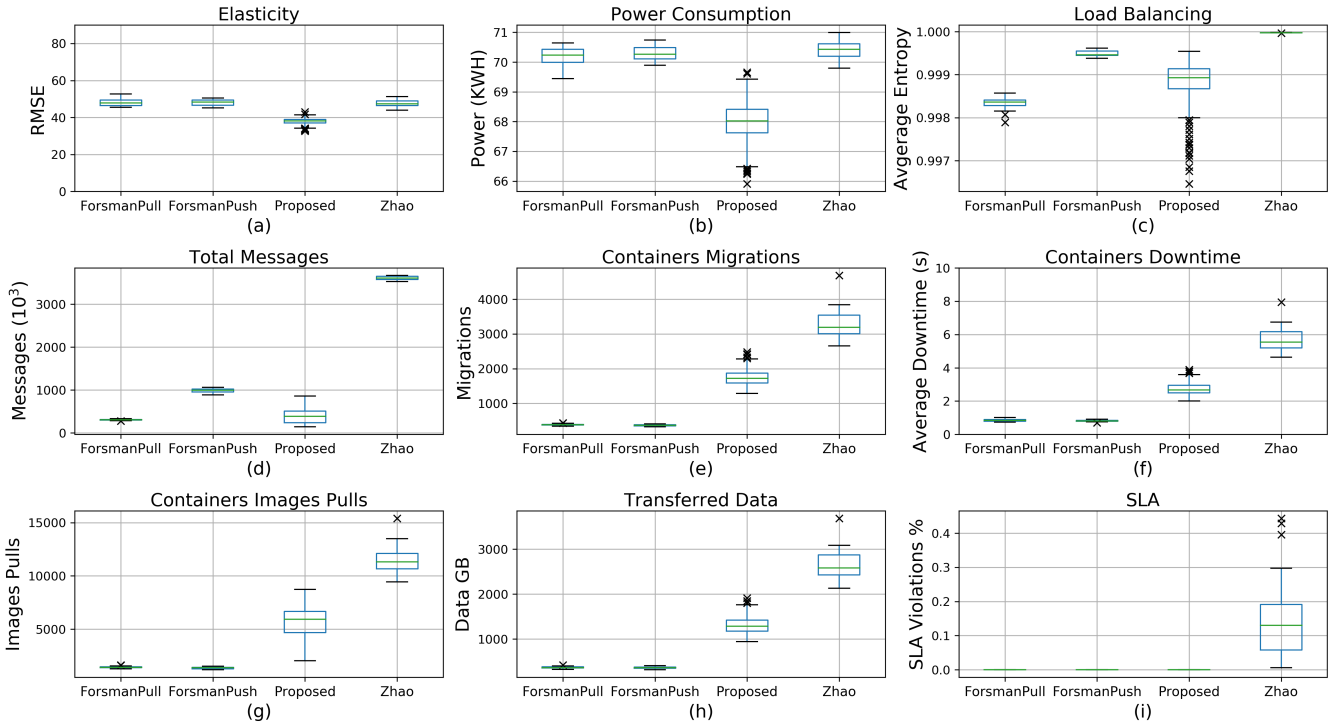

FIGURE 6. Algorithms comparisons (High-Utilization with Drain effect).

in the number of migrations performed and network metrics. Zhao compares and migrate policy maximizes its entropy but affects the number of messages, migrations, and network usage. Nevertheless, sometimes Zhao's experiences some SLA violation due to the usage of roulette-wheel selection algorithm that sometimes can select an overutilized host.

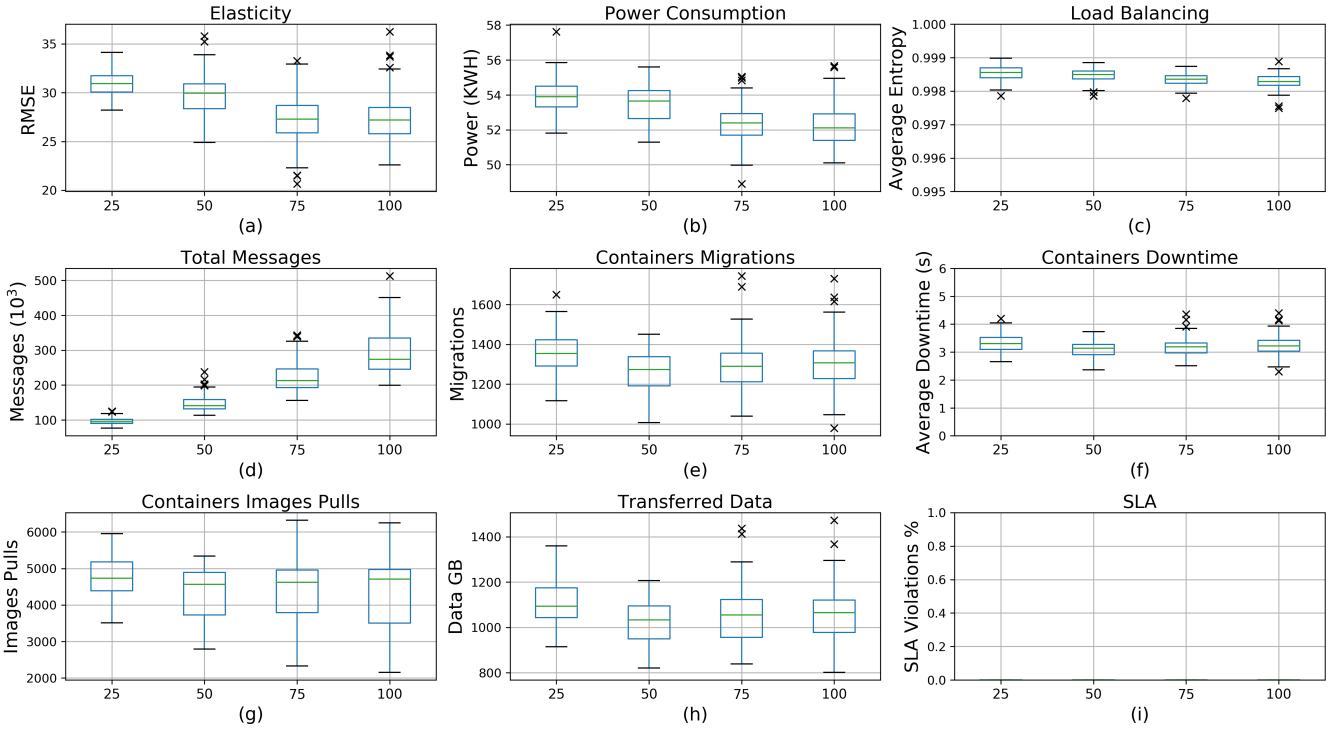

FIGURE 7. Effect of testing threshold (th) (mid-utilization with burst effect).

TABLE 4. Effect of selection policies in scenario 2 (high-utilization with drain effect).

Policy	Least Full	Least Pulls	Most Full	Random
Elasticity (RMSE)	37.86	38.01	38.06	38.24
Power (KWH)	67.81	67.99	67.90	67.93
Load Balancing (Average Entropy)	0.9982	0.9986	0.9986	0.9987
Containers Migrations	2006	1739	1735	1774
Containers Image Pulls	7165	3406	6216	6351
Total Message (10^3)	583	548	672	529
Transferred Data (GB)	1532	1192	1330	1356
Containers Average Downtime (sec)	3.10	2.69	2.70	2.74
SLA Violation %	0 %	0 %	0 %	0%

Finally, it shall be noted that the second scenario (High-Utilization with Drain effect) depicted in Figure 6 encounters quasi-conclusions.

C. EFFECT OF DIFFERENT SELECTION POLICIES

TABLE 3 shows the effect of host selection policy "HSP" on the proposed algorithm in the first scenario (Mid-Utilization with Burst effect) with a testing threshold "TH" equals 100%. Lest Full selection policy targets the host with the least utilization; therefore it outperforms its peers in elasticity and power consumption. However, it commits the largest number of migrations thus biggest downtime, total pulls and transferred data.

However, as expected, the least pulls excelled at the number of images pulls, the total number of messages, and total transferred data. The Most Full policies and exhibits the best performance in the migrations and consequently containers downtime. Nevertheless, it came worst in the elasticity, power consumption, and total messages. Finally, the SLA is not violated due to the nature of the scenario itself.

TABLE 4 the effect on the selection policies on the second scenario (70 % utilization with Drain effect) with a testing threshold (TH) equals 100%. As shown in TABLE 4 the policies behave the same way as the first scenario, but the results are proximate.

D. EFFECT OF TESTING THRESHOLD

Figure 7, describes the effect of the testing threshold (TT) on the performance metrics in the first scenario (Mid-Utilization with Burst effect) across all selection policies. As shown, the results of 75 and 100% thresholds are almost identical with the best achievable performance in almost all metrics

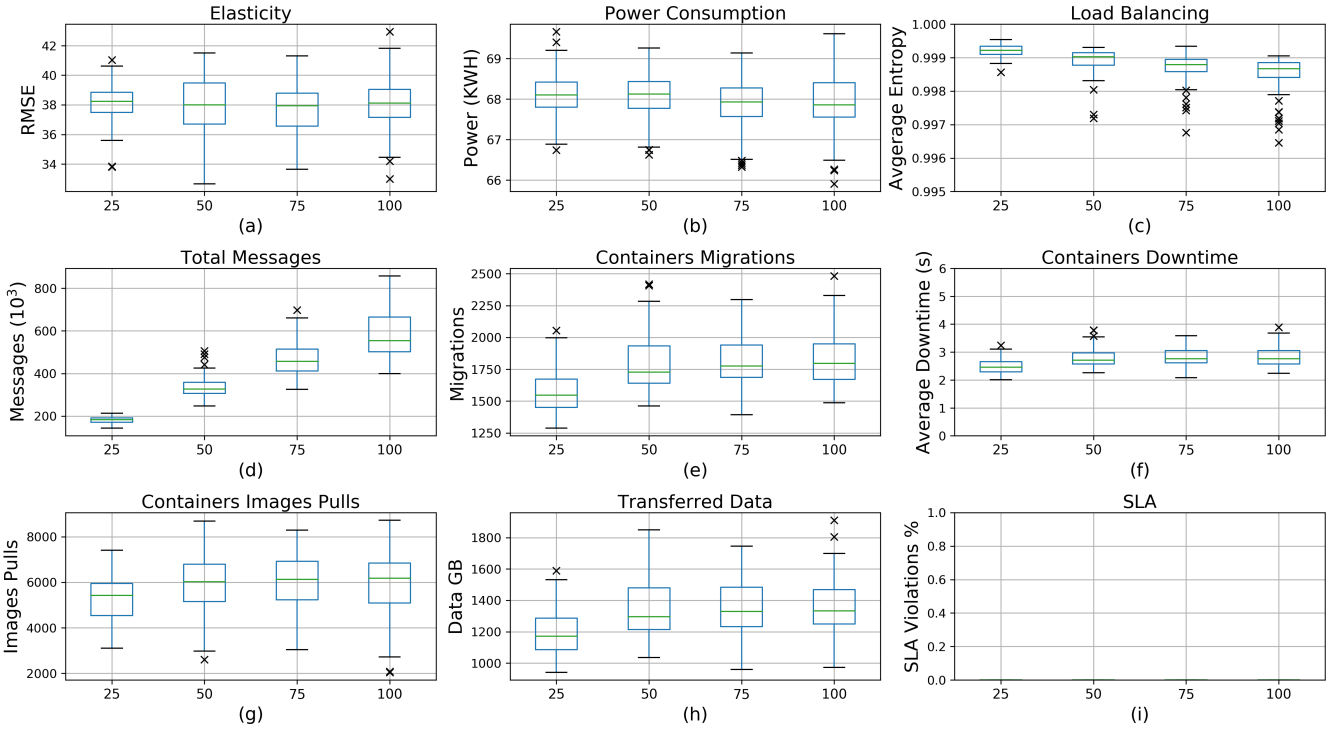

FIGURE 8. Effect of testing threshold (th)-(high-utilization with drain effect).

except for the total messages. Nevertheless, the 50% threshold is almost superior in the total migrations, containers downtime and transferred data. Finally, it shall be noted that the total messages increase linearly with the tested percentage without much of an increase in the other performance metrics. Moreover, the threshold effects are similar in High-Utilization with Drain effect as shown in Figure 8.

VI. CONCLUSION

This paper introduced a new infrastructure elasticity control algorithm for a containerized cloud. The proposed algorithm is boosting the load sharing methodology to include infrastructure elasticity. A series of experiments have been carried out to evaluate the performance of the proposed algorithm through elasticity centric metrics. The results demonstrated the algorithm capabilities to elasticate and handle flash crowds along with decreasing the management overhead and maintaining proximate load balancing. As demonstrated, the proposed algorithm is able to conserve 20% of the power at various utilization scenarios while keeping a balanced infrastructure. However, the elasticity process is based on containers' migrations which affect containers downtime, pulled images and migrated data.

In addition, the algorithms showed performance variability concerning the selection policy and testing threshold. For example, when selection policies are applied, the Least Full selection policy outperformance its peers in the elasticity and power consumption, while the least pull has a superior performance in network traffic minimization. Moreover, the results testing thresholds vindicated the advances of minimizing network traffic by different filtering methods without performance degradation

Finally, future work includes applying the proposed algorithm with different selection policies and to existent container management frameworks.

REFERENCES

[1] C. Pahl, A. Brogi, J. Soldani, and P. Jamshidi, "Cloud container technologies: A state-of-the-art review," *IEEE Trans. Cloud Comput.*, to be published.

[2] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Autonomic vertical elasticity of docker containers with ELASTICDOCKER," in *Proc. IEEE Int. Conf. Cloud Comput.*, Jun. 2017, pp. 472–479.

[3] *Cgroups*. Accessed: Jan. 19, 2018. [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v2.txt

[4] R. Dua, V. Kohli, S. Patil, and S. Patil, "Performance analysis of union and CoW file systems with docker," in *Proc. Int. Conf. Comput., Anal. Secur. Trends (CAST)*, 2016, pp. 550–555.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Philadelphia, PA, USA, 2015, pp. 171–172.

[6] C. Kan, "DoCloud: An elastic cloud platform for Web applications based on Docker," in *Proc. 18th Int. Conf. Adv. Commun. Technol. (ICACT)*, 2016, pp. 478–483.

[7] S. G. Saez, V. Andrikopoulos, R. J. Sanchez, F. Leymann, and J. Wettinger, "Dynamic tailoring and cloud-based deployment of containerized service middleware," in *Proc. IEEE 8th Int. Conf. Cloud Comput. (CLOUD)*, Jun. 2015, pp. 349–356.

[8] X. Wan, X. Guan, T. Wang, G. Bai, and B.-Y. Choi, "Application deployment using microservice and docker containers: Framework and optimization," *J. Netw. Comput. Appl.*, vol. 119, pp. 97–109, Oct. 2018.

[9] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical linux container migration," in *Proc. Global Internet Things Summit (GIoTS)*, 2017, pp. 1–4.
[10] R. Morabito, "Virtualization on Internet of Things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.
[11] D. Zhao, N. Mandagere, G. Alatorre, M. Mohamed, and H. Ludwig, "Toward locality-aware scheduling for containerized cloud services," in *Proc. IEEE Int. Conf. Big Data*, Nov. 2015, pp. 263–270.
[12] A. Giaretta, N. Dragoni, and M. Mazzara, "Joining jolie to docker: Orchestration of microservices on a containers-as-a-service layer," in *Proc. Adv. Intell. Syst. Comput.*, vol. 717, 2018, pp. 167–175.
[13] Y. Niu, F. Liu, X. Fei, and B. Li, "Handling flash deals with soft guarantee in hybrid cloud," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2017, pp. 1–9.
[14] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Trans. Services Comput.*, vol. 11, no. 2, pp. 430–447, Mar./Apr. 2018.
[15] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A framework and algorithm for energy efficient container consolidation in cloud data centers," in *Proc. 8th IEEE Int. Conf. Data Sci. Data Intensive Syst., 8th IEEE Int. Conf. Cyber, Phys. Social Comput., 11th IEEE Int. Conf. Green Comput. Commun.*, Dec. 2015, pp. 368–375.
[16] D. L. Eager, E. D. Lazowska, and J. Zahorjan, "Adaptive load sharing in homogeneous distributed systems," *IEEE Trans. Softw. Eng.*, vol. SE-12, no. 5, pp. 662–675, May 1986.
[17] M. Forsman, A. Glad, L. Lundberg, and D. Ilie, "Algorithms for automated live migration of virtual machines," *J. Syst. Softw.*, vol. 101, pp. 110–126, Mar. 2015.
[18] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *Proc. IEEE 4th Int. Conf. Cloud Comput. (CLOUD)*, Jul. 2011, pp. 500–507.
[19] M. Mathis, J. Semke, J. Mahdavi, and T. Ott, "The macroscopic behavior of the TCP congestion avoidance algorithm," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 3, pp. 67–82, 1997.
[20] *Part 3: Carrier Sense Multiple Access With Collision Detect on (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Standard 802.3, 2000.
[21] A. Beloglazov and R. Buyya, "Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers," *Concurrency Comput., Pract. Exper.*, vol. 24, no. 13, pp. 1397–1420, Sep. 2012.
[22] W. A. Hanafy. (2018). *Net Container Simulator*. [Online]. Available: https://github.com/washraf/.NetContainerSimulator
[23] C. Guo *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 139–152, 2015.
[24] Y. Zhao and W. Huang, "Adaptive distributed load balancing algorithm based on live migration of virtual machines in cloud," in *Proc. 5th Int. Joint Conf. (INC, IMS IDC)*, 2009, pp. 170–175.
[25] R. I. Dinita, G. Wilson, A. Winckles, M. Cirstea, and A. Jones, "Hardware loads and power consumption in cloud computing environments," in *Proc. IEEE Int. Conf. Ind. Technol.*, Feb. 2013, pp. 1291–1296.

WALID A. HANAFY received the B.Sc. and M.Sc. degrees in computer engineering from the Faculty of Engineering, Helwan University, in 2011 and 2018, respectively, where he is currently a Teaching Assistant, involved in many of its research projects. His current research interests include cloud computing, distributed systems, software architecture, software automation, and machine learning.

AMR E. MOHAMED received the Ph.D. degree in signal processing from Helwan University as a joint supervision (channel) grant with Connecticut University, Connecticut, USA, 2011. He is currently an Assistant Professor of electronics, communication, and computer engineering with the Faculty of Engineering, Helwan University. His research interests include signal processing, multi-agent cooperative systems, and cloud computing.

SAMEH A. SALEM received the B.Sc. and M.Sc. degrees in communications and electronics engineering from Helwan University, Helwan, Egypt, in 1998 and 2003, respectively, and Ph.D. degree in engineering from the Department of Electrical Engineering and Electronics, University of Liverpool, U.K., in 2008. In 2008, he was appointed as an Assistant Professor with the Department of Electronics, Communication, and Computer Engineering, Faculty of Engineering, Helwan University. He is also selected to be a Coordinator and an Academic Advisor with the Department of Communication and Information Technology, Uninettuno University, Italy, incorporation with the Faculty of Engineering, Helwan University. Furthermore, he is reviewing several proposals and research projects at the National Telecommunication Regulatory Authority (NTRA), Egypt. Moreover, he is the Postgraduate Coordinator between the Faculty of Engineering, Helwan University, and the Faculty of Engineering Sciences, Sinai University. In 2014, he was promoted to be an Associate Professor, and received an Honorary Research Fellow Position with the Department of Electrical Engineering and Electronics, University of Liverpool. He is currently a Consultant with the Egyptian Computer Emergency Response Team (EG-CERT) and the Head of Electronics, Communications, and Computer Engineering Department, Helwan University. His research interests include clustering algorithms, machine learning, data mining, parallel computing, and cloud computing.