The Journal of Systems and Software 155 (2019) 91–103

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

BrownoutCon: A software system based on brownout and containers for energy-efficient cloud computing

Minxian Xu[a,b,*], Rajkumar Buyya[b]

[a] Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
[b] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia

ARTICLE INFO

Article history:
Received 4 February 2019
Revised 16 May 2019
Accepted 17 May 2019
Available online 18 May 2019

Keywords:
Cloud data centers
Energy efficiency
Quality of service
Containers
Microservices
Brownout

ABSTRACT

VM consolidation and Dynamic Voltage Frequency Scaling approaches have been proved to be efficient to reduce energy consumption in cloud data centers. However, the existing approaches cannot function efficiently when the whole data center is overloaded. An approach called brownout has been proposed to solve the limitation, which dynamically deactivates or activates optional microservices or containers. In this paper, we propose a brownout-based software system for container-based clouds to handle overloads and reduce power consumption. We present its design and implementation based on Docker Swarm containers. The proposed system is integrated with existing Docker Swarm without the modification of their configurations. To demonstrate the potential of BrownoutCon software in offering energy-efficient services in brownout situation, we implemented several policies to manage containers and conducted experiments on French Grid'5000 cloud infrastructure. The results show the currently implemented policies in our software system can save about 10%–40% energy than the existing baselines while ensuring quality of services.

1. Introduction

Cloud computing has been regarded as a new paradigm for resource and service provisioning, which has offered vital benefits for IT industry by lowering operational costs and human expenses. However, the huge amount of energy consumption and carbon emissions resulted from cloud data centers have become a significant concern of researchers. Nowadays, data centers contain thousands of servers and their sizes range from 300–4500 square meters, which can consume more than 27,000 kWh energy per day (Mastelic et al., 2015). It is estimated that, in 2010, the energy consumption of data centers consumed 1.1% to 1.5% of total electricity worldwide (Mastelic et al., 2015). Moreover, the excessive usage of brown energy to generate power increases the carbon emission. It is also reported that about 2% carbon emissions of total carbon amount released into the atmosphere worldwide are from data centers (Lavallée, 2014). Recently, some dominant service providers have established a community, called Green Grid, to promote energy-efficient techniques to minimize the environmental impact of data centers (Beloglazov et al., 2012).

Unfortunately, reducing energy consumption is a challenging mission as applications and data are growing complex and consuming more computational resources (Liu et al., 2012). The applications and data are generally required to be processed within the required time, and to meet this requirement large and powerful servers are provisioned. To ensure the sustainability of future growth, cloud data centers are required to utilize the resource computing infrastructure efficiently and minimize energy consumption. To address this problem, the concept of green cloud was proposed, which aimed to reduce power consumption, energy cost, carbon emissions and also optimize renewable energy usage (Kong and Liu, 2015; Buyya et al., 2018). Therefore, in addition to resource provisioning and Quality of Service (QoS) assurance, data centers are required to be energy-efficient.

The dominant methods to improve resource utilization and reduce energy consumption are Virtual Machine (VM) consolidation (Beloglazov et al., 2012) and Dynamic Voltage Frequency Scaling (DVFS) (Kim et al., 2011). The VM consolidation method migrates VMs from underutilized hosts to minimize the number of active hosts, and the idle hosts are switched to low-power mode to save energy consumption. The DVFS method reduces energy usage by dynamically scaling voltage frequency. When the host is underutilized, the voltage frequency scales to a lower frequency to reduce power. These approaches have been proved to be efficient to save data center's power consumption, however, when the whole data

* Corresponding author.
E-mail addresses: minxianx@student.unimelb.edu.au (M. Xu), rbuyya@unimelb.edu.au (R. Buyya).

https://doi.org/10.1016/j.jss.2019.05.031
0164-1212/

center is overloaded, both of them cannot function efficiently. For example, the VMs can not be migrated if all the hosts are overloaded.

In data centers, another reason for high energy consumption lies in that computing resources are inefficiently utilized by applications. Thus, applications are currently built with microservice paradigm in order to utilize resources more efficiently. Microservice is referred as a set of self-contained application components. The components encapsulate their logic and expose their functionality via interfaces to enable flexible deployment and replacement. With microservices or components, developers and users can gain technological heterogeneity, resilience, scalability, ease of deployment, organizational alignment, composability and optimization for replaceability (Newman, 2015). In addition, microservices also brings the benefits of more fine-grained utilization control over the application resource.

To overcome the limitations of VM consolidation and DVFS, as well as improve the utilization of applications, we take advantage of brownout, a paradigm inspired from voltage shutdown to cope with emergency cases, in which the light bulbs emit fewer lights to save power (Xu and Buyya, 2019). Brownout is also applied to cloud scenarios, especially for microservices or application components that are allowed to be shortly deactivated to enhance system robustness. In brownout-compliant microservices, a control knob called dimmer is used to show the probability that whether a microservice should be executed or not (Klein et al., 2014). When requests are bursting and the system becomes overloaded, the brownout is triggered to temporally degrade the user experience, so that relieving the overloaded situation as well as saving energy consumption.

Microservices can be featured with brownout characteristic. An example of online shopping system with a recommendation engine is introduced in (Klein et al., 2014). The recommendation engine enhances the function of the system and increases profits via recommending products to users. However, because the engine is not the necessary component and it requires more resources in comparison to other components, it is not mandatory to keep running all the time, especially under the overloaded situation when requests have a long delay or even not served. Deactivating the engine enables service providers to serve more requests with essential requirements or QoS constraints. Apart from this example, brownout paradigm can also be applied to other systems that allow application components to be deactivated, especially for the container-based system that applications are built with microservice paradigm. With container technology, the microservices can be functionally isolated, thus the deactivation of some microservices will not influence other microservices. In addition, as microservices are light-weight, they can be deactivated/activated quickly to support the brownout approach.

In this paper, we propose and develop a software system, called *BrownoutCon*, which is inspired by brownout-based approach to deliver energy-efficient resource scheduling. The implementation of BrownoutCon is based on Docker Swarm (Docker, 2017) that provides the management of container cluster. The software system is designed and implemented as an add-on for Docker Swarm, which has no necessity to modify the configurations of Docker Swarm. The system also applies the public APIs of Grid'5000 (2017), which is a real testbed that provides power measurement for hosts. The aims of BrownoutCon are twofold: (1) providing an open-source software system based on brownout and Docker Swarm to manage containers; (2) offering an extensible software system for conducting research on reducing energy consumption and handling overloads in cloud data centers.

The BrownoutCon is designed and implemented by following the brownout enabled system model in our previous works (Xu et al., 2016; Xu and Buyya, 2017). Mandatory containers and optional containers are introduced in the system model, which are identified according to whether the containers can be temporarily deactivated or not. The brownout controller is the key part of the system model to manage brownout, which also provides the scheduling policies for containers. The problem of designing the brownout controller splits into several sub-problems:

1. Predicting the future workloads, so that the system can avoid overloads to foster the system robustness.
2. Determining whether a host is overloaded or not, so that the brownout controller will be triggered to relieve the overloads.
3. Deciding when to disable the containers, so that the system can relieve overloads and reduce energy consumption while ensuring QoS constraints.
4. Selecting the containers to be disabled, so that a better trade-off can be achieved between the reduced energy and QoS constraints.
5. Deciding when to turn the hosts on or into the low-power mode, so that the idle hosts can be switched into low-power mode to save power consumption.

Compared with VM consolidation approaches, the software system based on brownout and containers has two advantages: (1) a container can be stopped or restarted in seconds, while VM migration may take minutes. Thus, scheduling with containers is more light-weight and flexible than VMs. (2) the brownout-based approach provides another optional energy-efficient approach apart from VM consolidation and DVFS, which is also available to be combined with VM consolidation to achieve better energy efficiency, especially for the situation when the whole data center is overloaded.

To evaluate the proposed system in practice, we conduct our experiments on Grid'5000 (2017) real testbed. We also evaluate the performance of proposed system with real traces derived from Wikipedia[1] workloads.

The main *contributions* of our work are as follows:

- Proposed an effective system model that enables brownout approach to manage the containers and resources in a fine-grained manner;
- Designed and developed a software system based on Docker Swarm to provide energy-efficient approaches for cloud data centers;
- Experimental evaluations of our proposed software system on French Grid'5000 infrastructure for service providers to deploy microservices in an energy-efficient manner while ensuring QoS constraints.

The rest of the paper is organized as follows. Section 2 discusses the related work, followed by the system design and implementation in Section 3. Brownout-based policies implemented in BrownoutCon are presented in Section 4. In Section 5, we introduce our experiments setup and evaluate the performance of implemented policies under Grid'5000 testbed. Conclusions along with future work are presented in Section 6.

2. Related work

It is estimated that U.S. data centers will consume 140 billion kWh of electricity annually by the year 2020, which equals to the annual output of about 50 brown power plants that have high carbon emissions (Delforge, 2014; Bawden, 2016). To minimize the operational expenses and impacts on the environment, a variety of state-of-the-art works have been conducted to reduce data center energy consumption.

[1] See http://www.wikibench.eu/wiki/2007-10/ for more details.

There is a close relationship between resource utilization and energy consumption, as inefficient utilization of resource contributes to more power consumption (Kaur and Chana, 2015). Virtualization is an import technique in Clouds and it can improve resource utilization. Therefore, numerous energy-efficient resource scheduling approaches based on VM consolidation have been proposed. Consolidating VMs on fewer physical machines and turning the unused machines into the low-power mode reduce the number of active machines. Beloglazov et al. (2012) proposed several VM consolidation algorithms to save data center energy consumption. The VM consolidation process has been modeled as a bin-packing problem, where VMs are regarded as items and hosts are regarded as bins. The objective of these VM consolidation algorithms is mapping the VMs to hosts in an energy-efficient manner. Based on the VM consolidation approaches in this work, other works like (Belog1azov and Buyya, 2012; Chen et al., 2015; Han et al., 2016) have extended them to improve algorithm performance. Zhang et al. (2019) proposed VM allocation algorithm based on evolution algorithm to achieve energy efficiency in cloud data centers for reserved services. Experiments under both simulation and realistic environments showed that the proposed approach can effectively reduce energy consumption for a set of reserved VMs. Li et al. (2017) developed a Bayesian network-based estimation model for VM consolidation and took nine data center factors into consideration. The proposed approach can reduce energy consumption while ensuring QoS by avoiding inefficient VM migrations.

Another dominant approach to reduce energy consumption is Dynamic Voltage Frequency Scaling (DVFS). The DVFS approaches achieve energy reduction by adjusting frequencies of processors rather than using less active servers in VM consolidation. The DVFS approach investigates a trade-off between energy consumption and computing performance, where processors lower their frequency when they are lightly loaded and utilize full frequency when loads are heavy.

Kim et al. (2011) modeled real-time service as real-time VM requests, and proposed several DVFS algorithms to reduce energy consumption for the DVFS-enabled cluster. Arroba et al. (2015) proposed an approach combines DVFS and VM consolidation techniques by considering energy consumption and performance degradation together. Teng et al. (2016) presented several heuristic algorithms combining DVFS and VM consolidation together for batch-oriented scenarios. Fan et al. (2017) presented an online energy management approach by dynamically configuring voltage frequencies to minimize the power consumption for single processor scheduling while ensuring reliability requirement. All these approaches are focusing on developing algorithms for energy efficiency purposes.

Some research taking both energy consumption and QoS into account have been conducted, which is also the consideration of our proposed software prototype system. Dou et al. (2016) introduced an energy-aware dynamic VM scheduling approach for QoS enhancement in Clouds for big data, which aimed to benefit users with discount prices and reduce the execution time of tasks. Adhikary et al. (2017) developed a QoS-aware and energy-aware cloud resource management approach for multimedia applications, and proposed two distributed and localized resource management algorithms based on energy consumption and resource demands.

VM consolidation and DVFS have been proven to be efficient to reduce energy consumption in both theory and practice, however, both of them cannot function well when the whole data center is overloaded. Thus, brownout is applied to handle data center overloads and reduce energy consumption. Klein et al. (2014) applied brownout to design more robust applications under the overloaded or unpredicted situation. In our previous work, brownout was applied to save energy consumption in data centers. In Xu et al. (2016), we presented the brownout enabled system model and proposed several heuristic policies to find the microservices or application components that should be deactivated for energy saving purpose. The results showed that a trade-off existed between energy consumption and discount, and in Xu and Buyya (2017), we adopted approximate Markov Decision Process to improve the trade-off.

Compared to the existing energy-efficient approaches based on VMs, our software system is based on containers. Container technology is derived from the Linux LXC techniques (Bernstein, 2014), which provides mechanism to isolate processes on a shared operating system. Compared with VMs, containerization provides a fine-grained control on microservice resource usage and is more light-weight. Kozhirbayev and Sinnott (2017) compared several existing container-based technologies for Clouds and evaluated their strength and weakness. They concluded that containers can give almost the same performance of native systems. As the main reason of the energy consumption issue in clouds is due to the inefficient resource usage, and containers can provide a more fine-grained control on resources compared with VMs, we consider to apply container technology for energy efficiency purposes.

Currently, container technology is mostly focused on the orchestration of construction and deployment for containers (Pahl et al., 2017; Rodriguez and Buyya, 2018), and it has been applied for various purposes, such as scalability (Hightower et al., 2017), high availability (Naik, 2016), high utilization (Vavilapalli et al., 2013), high throughput (Schwarzkopf et al., 2013), and QoS (Boutin et al., 2014). For example, Liu et al. (2016) proposed a flexible container-based computing platform for scientific workflow. Baresi et al. (2016) introduced MicroCloud, which is a container-based solution for managing cloud resource efficiently. Santos et al. (2018) evaluated the energy consumption of different applications executed in Docker and bare metal. However, the energy efficient scheduling is not considered in these container-based work.

In our previous work (Xu et al., 2018), we have proposed an approach for managing energy in container-based clouds while focusing on scheduling algorithms design. Whereas, in this paper, we focus on the design and development of a new software system supporting brownout-based energy-efficient management of clouds.

Some of the software systems supported energy-efficient resource management in Clouds, including OpenStack Neat (Beloglazov and Buyya, 2015), Parasol (Goiri et al., 2013) and vGreen (Dhiman et al., 2009). However, none of them support brownout.

Table 1shows the comparison of related work. To the best of our knowledge, BrownoutCon is the first software system developed to reduce energy consumption with brownout based on containers, which also considers both energy consumption and QoS.

3. System architecture, design and implementation

The purpose of BrownoutCon is to provide a software system based on brownout and containers for energy-efficient cloud data centers. The system takes advantage of public APIs of Docker Swarm and is evaluated under Grid'5000 testbed. The system is designed to be extensible, which means new components can be added without the necessity to modify the original codes or configurations of Docker Swarm and Grid'5000.

Our software system is deployed on Docker Swarm master and worker nodes. Docker Swarm provides a platform for managing container cluster, monitoring status of swarm master and worker nodes, deploying containers on nodes, collecting resource usage of containers, controlling the lifecycle of containers, sending messages and commands between the master and worker nodes. Docker Swarm needs to be deployed on physical machines or virtual machines. Therefore, we adopt Grid'5000, a real testbed that

Table 1
Comparison of related work

Approach	Key Technique			Management Unit				Optimization Objective		Focus	
	DVFS	VM Consolidation	Brownout	Processor	Host	VM	Container	Energy	QoS/SLA	Algorithm Design	Software System
Beloglazov et al. (2012); Belog1azov and Buyya (2012)		√			√	√		√	√	√	
Chen et al. (2015)		√			√	√		√	√	√	
Han et al. (2016)		√			√	√		√	√	√	
Zhang et al. (2019)		√			√	√		√	√	√	
Li et al. (2017)		√			√	√		√	√	√	
Kim et al. (2011)	√			√				√	√	√	
Arroba et al. (2015)	√	√		√	√	√		√	√	√	
Teng et al. (2016)	√	√		√	√	√		√	√	√	
Fan et al. (2017)	√			√				√	√	√	
Dou et al. (2016)		√			√	√		√	√	√	
Adhikary et al. (2017)		√			√	√		√	√	√	
Klein et al. (2014)			√		√					√	
Xu et al. (2016) Xu and Buyya (2017)		√	√		√	√	√	√		√	
Liu et al. (2016)							√		√		√
Baresi et al. (2016)							√		√		√
Xu et al. (2018)			√		√		√	√	√	√	
Beloglazov and Buyya (2015)		√			√	√		√	√		√
Goiri et al. (2013)					√				√		√
BrownoutCon			√		√		√	√	√		√

provides access to ample resources for Docker Swarm deployment. We also take advantage of the Grid'5000 APIs to collect the energy consumption data of the machines. In the following sections, we discuss the system requirements, assumptions, system design and its implementation.

3.1. Requirements and assumptions

The components of the proposed software prototype system are running on the Docker Swarm master and worker nodes. Our current implementation assumes that a single instance of each key components is invoked on the master node, such as components for controlling brownout, monitoring system status, managing deployment policies and managing models. On each worker node, a single instance of a component that collects node information is running. When new nodes are joining the Docker Swarm as worker nodes, the master node is responsible for deploying containers to the nodes.

BrownoutCon saves the energy consumption and handles overloads via temporarily disabling some containers, therefore, we assume that the services in the target system (e.g. e-commerce system) are implemented with microservice paradigm and some services (e.g. recommendation engine service) are not necessary to keep running all the time.

The main optimization objective of our software system is reducing energy consumption, a precise power probe to collect energy usage is required. Container scheduling policies may use the energy usage data to make decisions on controlling containers.

Another requirement is that a manager is needed to control all the hosts to turn them into low-power mode or active. This manager is used by the brownout controller on master node to connect with other worker nodes via the communication protocol. Grid'5000 has provided the APIs to switch the status of hosts, and more details will be introduced in the following sections.

3.2. BrownoutCon architecture

Fig. 1 depicts the architecture of BrownoutCon, and the details of the main components are introduced as below:

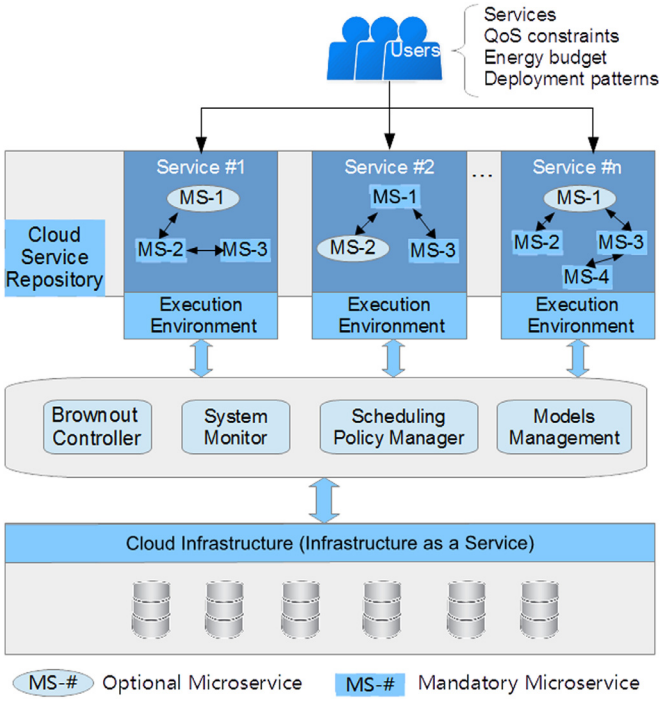

Fig. 1. BrownoutCon architecture.

(1) *Users:* This component contains user and requests information. It also captures system configurations such as predefined QoS constraints (e.g. average response time and SLA violations), energy budget and service deployment patterns according to users' demand.

(2) *Cloud service repository:* This component manages the services offered to users, including service information, such as service name and image. Each service may be constructed via a set of microservices. In order to manage microservices with brownout, the microservices are identified as mandatory or optional.

a. Mandatory microservices: These microservices keep running all the time when they are started and cannot be temporarily stopped, like database related microservices.

b. Optional microservices: These microservices can be deactivated temporarily depending on system status. Microservices are connected if there are communications between them. We consider that if one optional microservice is deactivated, then other connected microservices should also be deactivated.

Notes: A microservice can be identified as optional if the service/content it provides is defined as optional by its creators. For instance, the online recommendation engine in the online shopping system and the spell checker in the online editor system can be identified as optional microservices under resource constrained situations.

(3) *Execution environment:* This component provides the container-based environment for microservices or containers. The dominant execution environments for microservices or containers are Docker, Kubernetes, and Mesos. In BrownoutCon, we use Docker as the execution environment for microservices.

(4) *Brownout controller:* This component controls optional microservices or containers based on system status. It applies policies introduced in Section 4 to provide an efficient solution for managing brownout and containers. As noted in Section 1, brownout has a control knob called dimmer that represents the probability to execute microservices. We make some adjustments to make the dimmer of brownout to be adapted to this component as well as our architecture. Our dimmer is only applied to the optional microservices and its value is computed according to the severity of system overloads (the number of overloaded hosts in the data center).

(5) *System monitor:* It is a component that monitors the health of nodes and collects hosts resource consumption status. It uses the third-party toolkit to support its function, such as Grid'5000 public APIs that provide real-time data on infrastructure metrics, including host health, CPU utilization, and power consumption.

(6) *Scheduling policy manager:* This component provides and manages the policies for Brownout Controller to schedule microservices/containers. In order to ensure the energy budget and QoS constraints, different policies designed for different preferences are required. For instance, if the service provider wants to balance the trade-off between energy and QoS, then a policy that considers the trade-off is preferred.

(7) *Models management:* This component maintains the energy consumption and QoS models in the system. In BrownoutCon, the power consumption model is closely related to the utilization of microservice or container, and the QoS model is applied to define the QoS constraints.

(8) *Cloud infrastructure:* Under Infrastructure as a Service model, it is a component that offers physical resources to users, where microservices or containers are deployed. In our experiments, we use Grid'5000 as our infrastructure. More details are given in Section 5.

3.3. Energy-efficient scheduling architecture

The main purpose of our software system is energy efficiency, and the main approach to achieve this goal is through energy-efficient scheduling policies. Deriving from BrownoutCon architecture, Fig. 2 shows the energy-efficient scheduling architecture based on brownout, which depicts the BrownoutCon from the energy-efficient scheduling perspective.

In this scheduling architecture, clients submit their requests to the system, and Docker Swarm Manager dispatches the requests to containers and hosts. The System Monitors collect the energy and utilization information from hosts, and then send the information to Brownout Controller. With the information from System Monitors, the Brownout Controller refers to the host power consumption or utilization models to compute how much utilization/energy should be reduced. Then the Brownout Controller makes decisions based on scheduling policies to switch the states of hosts and containers, such as turning the hosts into low-power mode or deactivating containers.

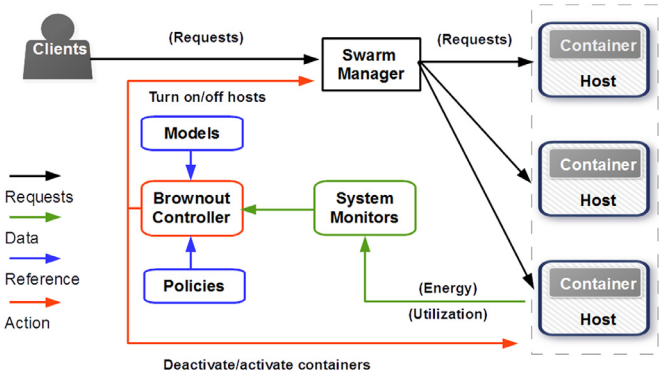

Fig. 2. Energy-efficient scheduling architecture.

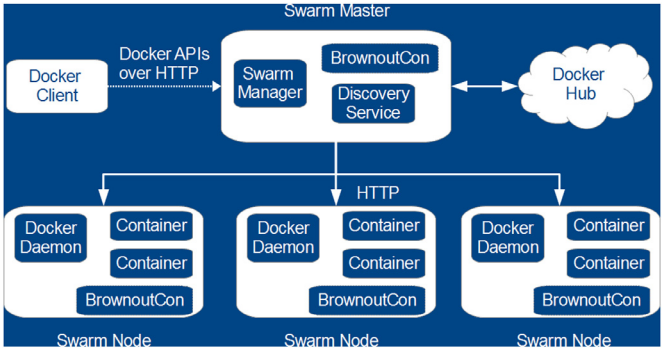

Fig. 3. BrownoutCon integrated with Docker Swarm.

3.4. Integration with Docker Swarm

BrownoutCon is installed on Docker Swarm node independently of Docker Swarm services. In addition, the activities of BrownoutCon are transparent to the Docker Swarm services, which means Docker Swarm does not need to reconfigure to fit with BrownoutCon and use its brownout feature. In other words, BrownoutCon can be installed on existing Docker Swarm cluster without modifying the configurations.

BrownoutCon achieves the transparency via the interactions with the public APIs of Docker Swarm cluster. BrownoutCon uses the APIs to obtain information about containers deployment, containers utilization, and containers properties. Although the operations of BrownoutCon will affect the system status and containers state by deactivating or activating containers, it is transparently processed by Docker Swarm public APIs.

The implication of this integration approach represents that the container deployment is handled by Docker Swarm, and BrownoutCon makes decisions on deactivation or activation of containers. Fig. 3 shows how BrownoutCon is integrated into Docker Swarm. In Docker Swarm, the nodes are categorized as two classes: swarm master node and swarm worker node. The master node is responsible for maintaining cluster state, scheduling services (containers) and serving swarm mode with Docker APIs over HTTP, while the purpose of worker nodes is executing containers. The respective BrownoutCon components are deployed on master and worker nodes.

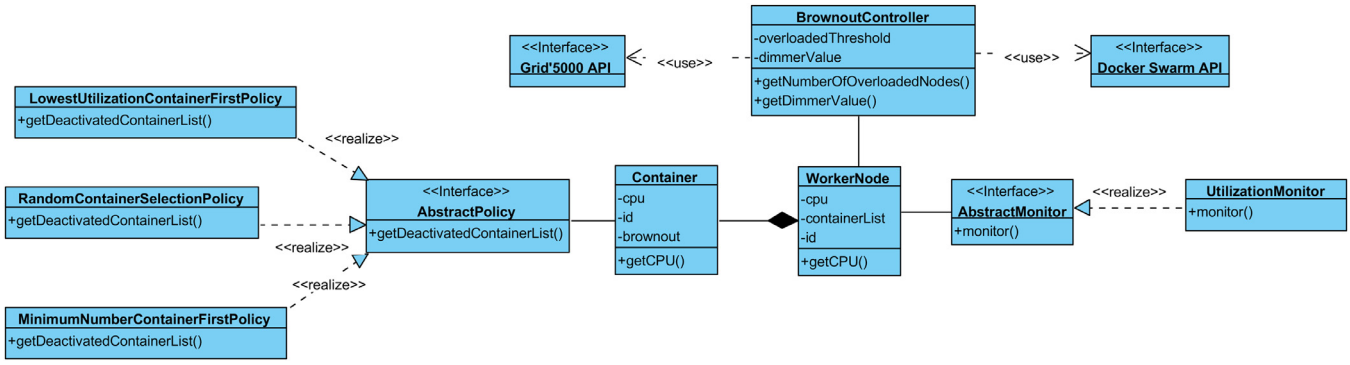

Fig. 4. Entity interactions in BrownoutCon.

3.5. Containers deployment with compose file

Docker provides compose[2] tool to deploy multiple containers, in which a configuration file is used to configure containers properties. With the compose file, the containers can be easily deployed and managed on clusters. In the compose file of our web application, to identify the recommendation engine microservice as optional, we labeled it as optional in the brownout feature. Moreover, as previously mentioned, the optional containers are only allowed to be deployed on the worker node, thus, we configure the placement constraint of this microservice as the worker node. More deployment properties can also be configured in the compose file.

3.6. Entity interaction diagram

To implement the aforementioned architecture and functionalities, we use Java to develop our software system. The main classes of BrownoutCon are depicted in Fig. 4. The details of these classes are as below:

Docker Swarm API: This class wraps Docker Swam APIs and provides the interface for BrownoutController class to call. The Docker Swarm APIs offer the functions to fetch the information of containers and operate on containers, such as collecting containers utilization, containers id, containers property (optional or mandatory), deploying and updating containers with the compose file, deactivating and activating containers.

Grid'5000 API: This class uses Grid'5000 APIs to collect hosts energy consumption and switch status of hosts. Grid'5000 provides APIs to gather the total power at per second rate for all the hosts in data center. The APIs also allow BrownoutController class to switch the hosts into low power mode or turn the hosts on.

WorkerNode: This class models the host in the data center. Attributes of a WorkerNode include the CPU utilization and the containers deployed on the host. To be consistent with the status of real hosts, when the software system is running, the WorkerNode instances will keep updating their CPU utilization and container lists.

AbstractMonitor: It provides an interface to monitor the status system. With the monitored information, the system can know how many hosts are overloaded and make decisions based on this information. Other monitors, such as memory or network monitors can be extended if they implement the AbstractMonitor.

Container: The Container class models the containers deployed on hosts. The class defines the basic information of containers, including container id, CPU utilization and other information that can be fetched via Docker Swarm APIs.

AbtractPolicy: It is an interface that defines the functions that scheduling policies should implement. To deactivate some containers temporarily and reduce energy consumption, the policies that implement the AbstractPolicy interface are responsible for finding the containers that should be deactivated. The details of our implemented policies in BrownoutCon will be introduced in Section 4.

BrownoutController: This class is the core class of our software system. It assembles all the information from different sources and makes the decision for controlling hosts and the containers on them. BrownoutController knows system status from Docker Swarm APIs, Grid'5000 APIs and WorkerNode instances, and triggers brownout to handle overloads and reduce energy consumption via operations on hosts or containers.

3.7. Sequence diagram

To provide an in-depth understanding of the working process of BrownoutCon, Fig. 5 shows a sequence diagram of handling requests by our software system. Firstly, the users submit their requests to a web application called Weave Shop (more details about this application will be introduced in Section 5.2) Then the Weave Shop sends the information of requests to BrownoutCon, and BrownoutCon keeps collecting nodes and containers information periodically via Grid'5000 and Docker Swam public APIs, respectively. When BrownoutCon is collecting information, if the system is overloaded, which means the Weave Shop cannot handle all the incoming requests, the BrownoutCon adds nodes to serve requests. The BrownoutCon also triggers brownout-based policies to deactivate containers to relieve overloads and reduce energy consumption. After these operations, the information of the nodes and containers are updated. Once the system is not overloaded, BrownoutCon activates the containers or removes the nodes from active nodes list (switching nodes into low-power mode). Upon the completion of operations on containers and nodes, the updated information is sent to BrownoutCon.

4. Policies implemented in BrownoutCon

To demonstrate BrownoutCon software system capability, we plugged/incorporated some policies originally evaluated by simulations in Xu et al. (2016). As noted in Section 1, the scheduling problem can be divided into several sub-problems: (1) workload prediction; (2) overloaded status detection (3) brownout trigger; (4) deactivated containers selection; and (5) hosts scaling. In this section, we will introduce the implemented policies for reference. It is noted that the introduced policies are not the main focus of this paper. The focus of this work is designing and implementing the software system based on brownout and containers.

[2] See https://docs.docker.com/compose/compose-file/ for more details.

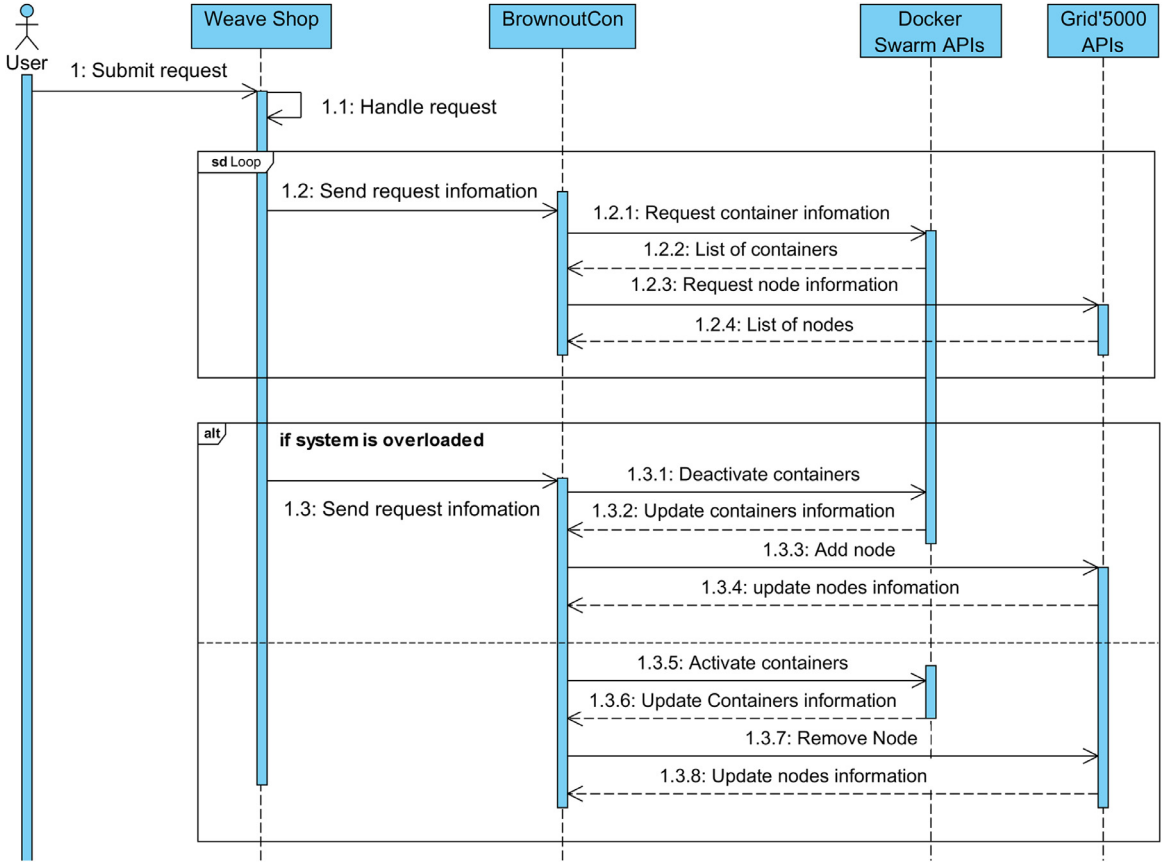

Fig. 5. Sequence diagram of handling requests by BrownoutCon.

4.1. Workload prediction

We apply workload prediction to avoid overloads and improve system robustness. To predict the future workloads based on the previous workloads, we adopt the sliding windows as presented in Algorithm 1. The motivation of sliding windows is giving more weights to the request rates of recent time intervals. Let L_w to be the window size that is a constant integer value, e.g. 5, $num(k)$ to be the actual number of requests at time interval k, we estimate the number of requests at the time interval t as the average number of requests in the previous L_w windows as shown in Eq. (1). To ensure enough historical data to be used for prediction, the time interval t for requests prediction should be no less than the window size L_w, for instance, if the window size $L_w = 5$, the time interval for requests predication can start from 5. The predicted number of requests $\hat{num}(5)$ at time interval 5 equals to the average number of actual requests of $num(0), num(1), \ldots, num(4)$. The sliding window is moving forward along with the time.

Algorithm 1 Algorithm for predicting future workload based on sliding windows.

Input: sliding window size L_w, the number of requests at previous L_w time intervals, the predicted time interval t ($t \geq L_w$)
Output: the predicted number of requests $\hat{num}(t)$ at time interval t

1: **for** k from $t - L_w$ to $t - 1$ **do**
2: $\quad \hat{num}(k+1) \leftarrow \hat{num}(k) + num(k)$
3: **end for**
4: $\hat{num}(t) \leftarrow \hat{num}(k+1)/L_w$
5: **return** $\hat{num}(t)$

Based on the predicted workloads, the number of active hosts can be dynamically scaled in and out, which will be introduced in Section 4.5. The performance of Algorithm 1 will be evaluated in Section 5.1.

$$\hat{num}(t) = \frac{1}{L_w} \sum_{k=t-L_w}^{t-1} num(k) \tag{1}$$

4.2. Overloaded host detection

In our experiments, we use a predefined overloaded threshold to detect whether a host is overloaded or not. For instance, if the overloaded threshold is defined as 85%, the host is regarded as overloaded when its CPU utilization is above 85%. Currently, we only adopt CPU utilization to detect the overloaded host.

Eqs. (2)and (3) show the way to calculate the number of the overloaded host. We use n_i^o to denote whether host i is overloaded or not, which is detected by the utilization u_i and overloaded threshold T_u. If u_i is no less than T_u, n_i^o equals to 1, otherwise it equals to 0. The total number of the overloaded host is denoted as n_o, which is the sum of n_i^o for all the hosts.

$$n_i^o = \begin{cases} 1, & if\ u_i \geq T_u \\ 0, & if\ u_i < T_u \end{cases} \tag{2}$$

$$n_o = \sum_{i=0}^{n-1} n_i^o \tag{3}$$

4.3. Brownout trigger

Once there are hosts detected as overloaded, the brownout mechanism will be triggered to handle the overloads as well as

to reduce energy consumption. As noted in Section 1, firstly, the algorithm is required to calculate the dimmer value, which is the control knob to represent the probability to trigger brownout on hosts. The dimmer value θ_t at time t is calculated based on the number of overloaded hosts n_o as shown in Eq. (4):

$$\theta_t = \sqrt{n_o/n} \tag{4}$$

Then the algorithm computes the expected utilization reduction on overloaded hosts. The expected utilization reduction u_i^r of host i is the product of dimmer value θ_t and the host utilization u_i as:

$$u_i^r = \theta_t \times u_i \tag{5}$$

4.4. Deactivated containers selection

Based on the expected utilization reduction, the policies select containers to deactivate based on different containers selection policies. In BrownoutCon, we have implemented three containers selection policies for deactivation. Based on the strategy design pattern[3], these policies implement the AbstractPolicy interface in Fig. 4 and can be selected independently at runtime.

4.4.1. Lowest utilization container first policy

The Lowest Utilization Container First (LUCF) policy selects a set of containers to reduce the utilization of overloaded hosts. The objective of LUCF is that the utilization after reduction is expected to be less than the overloaded threshold, and the difference between the expected utilization reduction and the sum of deactivated containers utilization is minimized. Thus, the host utilization is reduced and the reduced utilization is close to the expected reduction. The deactivated container list is defined in Eq. (6). We use $u_i^{'}$ to denote the utilization of host i after the containers in the deactivated lists are deactivated, which equals to $u_i - u_i^{dcl}$. The utilization of all the containers in dcl_i is denoted as u_i^{dcl}. The $\min(|u_i^r - u_i^{dcl}|)$ represents the to minimize the absolute value of $u_i - u_i^{dcl}$.

$$dcl_i = \begin{cases} \{u_i^{'} \le T_u, \min(|u_i^r - u_i^{dcl}|)\}, & if\ u_i \ge T_u \\ \emptyset, & if\ u_i < T_u \end{cases} \tag{6}$$

Algorithm 2 presents the pseudocode of LUCF. The LUCF sorts the optional containers list ocl_i based on container utilization in ascending order so that the container with the lowest utilization is at the head of the list. The size of ocl_i is $ocl_i.size()$. The algorithm checks the hosts one by one, if the first container c_0 on host i has the utilization greater than u_i^r, c_0 is put into the deactivated container list dcl_i. Since we consider connected microservices, the policy also adds the container's connection tag T_0^c (a string value) that indicates how it is connected with other containers into a set S for recording connections. However, if the utilization of the first container is less than the expected utilization reduction, LUCF finds a containers sublist to deactivate more containers. The sublist is the one that has the sum of utilization that is closest to the expected utilization reduction than other sublists. Same as previous operations, these containers are put into the deactivated container list dcl_i and their connection tags are put into the set S. Then, the algorithm finds other connected containers and put them into the deactivated container list.

4.4.2. Minimum number of containers first policy

As formalized in Eq. (7), in order to deactivate fewer containers so that more optional functionalities can be provided, we also implement Minimum Number Containers First (MNCF) policy, which selects the minimum number of containers while saving the power consumption. Since it is quite similar to the LUCF, the pseudocode of MNCF is not provided here. The $\min(dcl_i.size())$ represents the objective to minimize the size of the deactivated container list.

$$dcl_i = \begin{cases} \{u_i^{'} \le T_u, \min(dcl_i.size())\}, & if\ u_i \ge T_u \\ \emptyset, & if\ u_i < T_u \end{cases} \tag{7}$$

4.4.3. Random container selection policy

Based on a uniformly distributed discrete random variable X that selects a subset of dcl_i randomly, the Random Container Selection (RCS) policy uses uniform distribution function $U(0, ocl_i.size() - 1)\}$ to randomly select a number of optional containers to reduce energy consumption, as presented in Eq. (8).

$$dcl_i = \begin{cases} \{u_i^{'} \le T_u, X = U(0, ocl_i.size() - 1)\}, & if\ u_i \ge T_u \\ \emptyset, & if\ u_i < T_u \end{cases} \tag{8}$$

Algorithm 2 Lowest Utilization Container First policy (LUCF).

Input: the number of hosts n in data center, overloaded threshold T_u, deactivated container list dcl_i on host i, the optional container list ocl_i of host i, which is sorted based on utilization of containers u_j^c in ascending order, deactivated tag set S, connection tag T_j^c of container c_j

Output: deactivated container list dcl_i

```
for i ← 0 to n − 1 do
  if u_i > T_u then
    if u_0^c ≥ u_i^r then
      add c_0 into dcl_i
      add T_0^c into S
    end if
    for c_j in ocl_i (j = 0, 1, 2, ..., ocl_i.size() − 1) do
      if (u_j^c ≤ u_i^r) & (u_i^dcl ≤ u_i^r) then
        add c_j into dcl_i
        add T_j^c into S
        minimize |u_i^r − u_i^dcl|
      end if
    end for
    for c_j in ocl_i (j = 0, 1, 2, ..., ocl_i.size() − 1) do
      if T_j^c in S then
        add c_j into dcl_i
      end if
    end for
  end if
  deactivate containers in dcl_i
end for
return dcl_i
```

4.5. Hosts scaling

To scale the number of active hosts, we adopt the hosts scaling algorithm in (Toosi et al., 2017) as shown in Algorithm 3, which is a predefined threshold-based approach. With profiling experiments, we set the overloaded requests threshold as the number of requests when the host cannot respond within an acceptable time limit. The algorithm computes the required hosts as the predicted number of request divided by the profiling number of requests of the overloaded threshold. If the required number of hosts is more than current active hosts, more hosts will be added to provide services, otherwise, if current active hosts are adequate, then the excess machines can be set as low-power mode to save energy consumption.

5. Performance evaluation

In this section, we evaluate our proposed software prototype system by conducting experiments under Grid'5000 infrastructure.

[3] See https://en.wikipedia.org/wiki/Strategy_pattern for more details.

Algorithm 3 Hosts scaling algorithm.

Input: number of hosts n in data center, number of active hosts n_a, number of requests when host is overloaded num_{T_u}, predicted number of requests $n\hat{u}m(t)$ at time t.
Output: number of active hosts n_a

```
n_a ← ⌈nûm(t) ÷ num_{T_u}⌉
n' ← n_a − n
if n' > 0 then
    Add n' hosts
else if n' < 0 then
    Remove |n'| hosts
else
    no host scaling
end if
return n_a
```

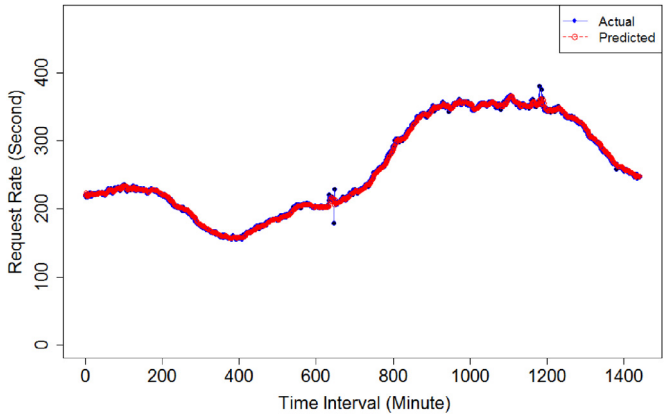

Fig. 6. Requests rate of Wikipedia trace (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.).

The goals of this section include: (1) evaluating the behavior of the software system in an experimental environment, and (2) demonstrating suitability of the proposed system to enable experimental evaluations and scheduling policies in a practical setting.

5.1. Workload traces

To make the experiments reproducible, we use the real trace from Wikipedia requests on October 17, 2017 to replay the workload of Wikipedia users. The trace includes data on requests time stamp and their accessed pages. We filter[4] the requests based on per second rate and generate the requests rate. The original request rate is around 1500–3000 per second. To scale the workload set to fit with our experiments, we use 10% of the original user requests size. Fig. 6 shows the requests rate per second during the day. The blue line is the actual trace derived from Wikipedia and the red line is the predicted trace based on the sliding window (sliding window size is 5) as introduced in Section 4. We can observe some anomalies during intervals 600–800, which can be due to the unpredicted network congestion. While during most time intervals, the variances between actual trace and predicted trace are small.

We conduct statistical analysis for the actual and predicted traces with Root Mean Square Error (RMSE) metric, which has been widely used in statistical analysis to verify experimental results and measures the average spread of errors as $RMSE = \sqrt{\frac{1}{S}\sum_{s=1}^{S}(\hat{I}_s - I_s)^2}$, where S is the size of the trace, $\hat{I}_s$ is the predicted value and I_s is the actual value. If RMSE is small, it means the predicted values are close to the actual values. In our experiments, the $RMSE = 2.67$, which represents the predicted trace is close to the actual one, and the predicted trace can serve a guide for host scaling strategy.

5.2. Application deployment

We use the Weave Shop[5] web application that implemented with containers as the application in our scenario. The Weave Shop is a shopping system for selling socks online and has multiple microservices, including user microservice to handle user login, user database microservice for user information storage, payment microservice to process transactions, font-end microservice to show the user interface, catalog microservice for managing item for sale and etc. As these microservices are implemented independently, they can be deployed and controlled without impacting other microservices. The application is deployed by the compose file as introduced in Section 3.5, and part of the microservices are configured as optional, e.g. recommendation engine is noted as optional.

The user interface may be influenced due to the deactivation of some microservices. Fig. 7 shows the user interface of Weave Shop application. Fig. 7(a) is the user interface when full services are provided during no resource saturated scenario, while Fig. 7(b) illustrates the user interface when brownout is triggered and the recommendation engine service/container is deactivated. As a result, other recommended products are not showed in Fig. 7(b).

5.3. Performance metrics

We adopt total energy consumption, average response time and SLA violation ratio as our performance metrics, and their definitions are as follows:

Total energy consumption: The total energy consumption represents the amount of energy that is consumed by the software system. The key reasons for adopting this metric are: (1) one of the objectives of BrownoutCon is reducing energy consumption, and (2) this metric is widely used in research articles for energy efficient clouds. The total energy consumption $E(t)$ during time interval t is formed as the sum of all the host energy consumption in the data center as shown in Eq. (9). Here, we only care about the physical server's energy consumption rather than other network devices or cooling equipment.

$$E(t) = \sum_{i=0}^{n-1} \int_{t}^{t+1} P_i(t)dt, \tag{9}$$

where n is the total number of hosts in the data center, and $P_i(t)$ is the power at time t of host i. And the total energy consumption of data center during observation time period T can be represented as $\sum_{t=0}^{T} E(t)$. The Grid'5000 cluster provides APIs for the fine-grained power measurement of each physical server at per second rate. We use the APIs to collect the power measurement data (in watts unit) during our observation period and calculate the power consumption (in kWh unit).

Average response time: This metric measures the time between users send requests and receive the response on average. We choose this metric because (1) another objective of BrownoutCon is ensuring QoS, and (2) the average response time is a widely used metric of QoS. And the average response time is a widely used metric of QoS. We can use this metric to quantify the overall QoS of the system. We aim to ensure this metric to be below a specific value, e.g. 1 s.

[4] The details about how we filter the raw data are provided at: https://github.com/Cloudslab/BrownoutCon.

[5] See https://github.com/microservices-demo/microservices-demo for more details.

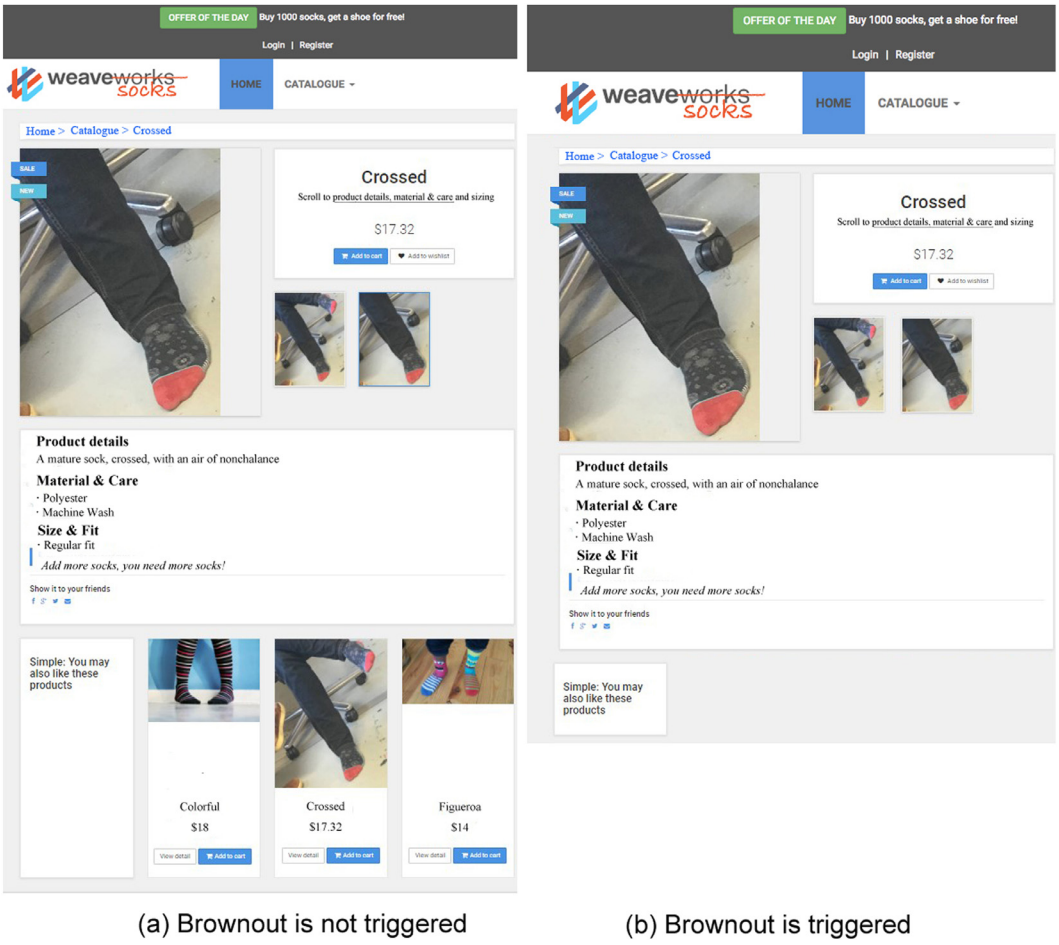

Fig. 7. Impact on user interface when brownout is triggered.

SLA violation ratio: As another metric to measure QoS, this metric quantifies the ratio of requests that fail to satisfy the predefined SLA. The reason for choosing this metric is the same as the reason for using average response time as metric. The metric is formalized as:

$$SVR = \frac{num_{err}}{num_a}, \tag{10}$$

where num_a is the total number of requests sent to the system, and num_{err} is the number of requests failing to get the response. *SVR* should also be optimized to be lower than a predefined value, e.g. 1%.

5.4. Experimental testbed

The testbed we used for evaluation is Grid'5000, and we adopt the cluster equipped with power measurement APIs at Lyon site. The hardware specifications are as below:

- 11 × Sun Fire V20z[6] with AMD Opteron 250 CPU (2 cores, 2.4 GHz) and 2 GB memory, and all the hosts are running on Debian Linux operating system.

We choose the machines in the same site so that we can reduce the network influence of uncontrolled traffics from other sites. Among the 11 servers, nine are running as the Docker Swarm worker nodes, one is running as the Docker Swarm master node, and another node contains workload trace and installed with JMeter[7] for sending requests to Docker Swarm cluster. The energy consumption of the workload trace node is not counted, as it is not regarded as a part of the cluster to provide services. All required softwares, such as Docker, Java, Ansible[8] and JMeter have been installed on these machines to minimize the effects of CPU utilization and network delay.

5.5. Experimental design and setup

In our experiments, the overloaded threshold is configured as 85%, as this value has been evaluated in our previous work Xu et al. (2018) that it can achieve a better trade-off between energy consumption and QoS than other values, e.g. below 80% or above 90%, as small overloaded threshold triggers brownout too frequently and large overloaded threshold will not trigger brownout. Another configured parameter is the optional utilization percentage, which represents how much CPU utilization is allowed to be given to the optional containers. According to (Xu et al., 2016), the change of this parameter has an impact on energy consumption. We vary this parameter with 10%, 20%, and 30% respectively, as the large values, like 40%, can lead to non-negligible performance degradation, and the small values, like 5%, cannot reduce energy consumption effectively. The experiments are conducted by going through the algorithms as below:

[6] The maximum power of this model is 237 Watts, and the sleep mode consumes 10 Watts.

[7] See http://jmeter.apache.org/ for more details.

[8] See https://www.ansible.com/ for more details.

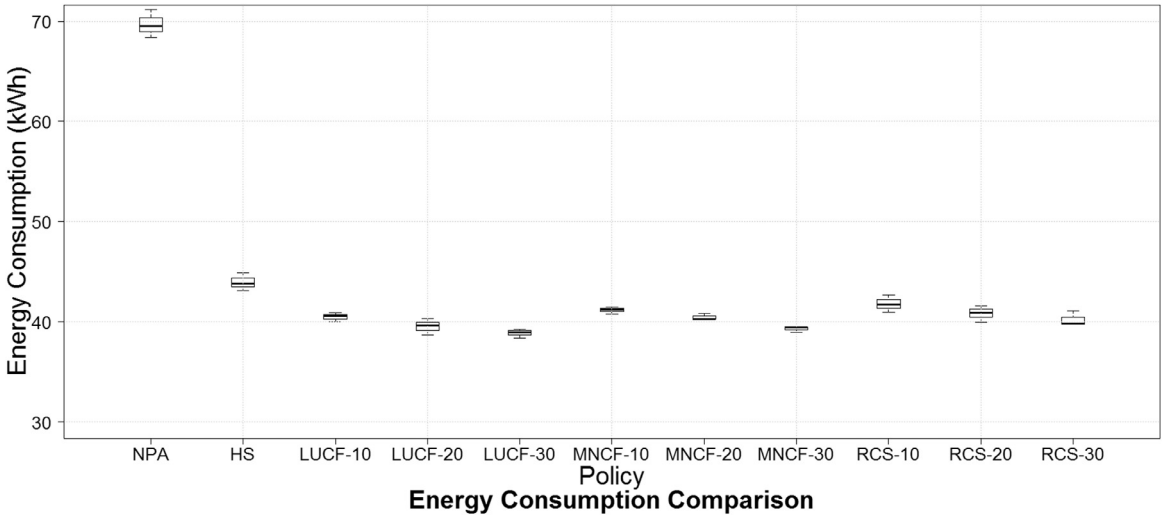

Fig. 8. Energy consumption comparison.

1. NPA algorithm (Beloglazov et al., 2012) — A baseline algorithm that does not consider overloads and optional containers, where the hosts are running all the time and containers are not deactivated.
2. HS (Toosi et al., 2017) — Another baseline algorithm that applies the host scaling algorithm in Algorithm 3, while not applying brownout-based policies.
3. The LUCF, MNCF and RCS algorithms introduced in Section 4 are with the varied optional utilization percentages from 10% to 30% in increments of 10%.

We evaluate the energy consumption, average response time and SLA violation ratio for these algorithms. We run each experiment 5 times to deal with the variance resulted from random factors, such as initial containers deployment, network latency, and application running status.

5.6. Experimental results and analysis

Fig. 8 depicts the energy consumption comparison of different algorithms. From the results, NPA has the highest energy consumption with 69.6 kWh, and HS reduces it to 43.9 kWh. For the brownout-based algorithms with varied parameters, the energy consumption of LUCF is from 40.5 kWh to 38.8 kWh when the optional utilization percentage is increased from 10% to 30%. For the MNCF, its energy is close to LUCF, which ranges from 41.1 kWh to 39.2 kWh when optional utilization percentage is varied. As for the RCS, it decreases the energy from 41.4 kWh to 38.8 kWh. All the brownout-based algorithm have shown a significant reduction around 40% to 44% in energy consumption than NPA, and they can also save about 6% to 12% power consumption than HS with different parameter settings.

We also compare the average response time in Fig. 9. Although NPA consumes more energy than other algorithms, with the adequate resources, the average response time is the lowest as 174.3 ms. The average response time of HS is 611.6ms, while the other brownout-based algorithms decrease this value. LUCF lowers the average response time from 472.6 ms to 425 ms, MNCF reduces it from 485.6 ms and reaches 427.3 ms with 30% optional utilization percentage, and the average response time of RCS ranges from 564.0 ms–511.3 ms.

In Fig. 10, the SLA violation ratios are compared. As NPA has enough resources, it does not experience any SLA violation, while in HS, it has 4.3% SLA violation. Compared with HS, LUCF relieves the SLA violated situation, reducing it from 2.1% to 0.5%. Similar to LUCF, MNCF also decreases the SLA violation to 0.5% from 2.3% when more optional utilization percentage is allowed. As for RCS, its SLA violation drops from 3.2%–0.9%.

Table 2
The experiment results.

Algorithm	Energy	Avg. Response Time	SLAVR
NPA	69.6 kWh	**174.3 ms**	-
HS	43.9 kWh	611.6 ms	4.3%
LUCF-10	40.5 kWh	472.6 ms	2.1%
LUCF-20	39.5 kWh	470.3 ms	1.4%
LUCF-30	**38.8 kWh**	425.0 ms	**0.5%**
MNCF-10	41.1 kWh	485.6 ms	2.3%
MNCF-20	40.4 kWh	471.3 ms	1.4%
MNCF-30	39.2 kWh	427.3 ms	0.5%
RCS-10	41.4 kWh	564.0 ms	3.2%
RCS-20	39.8 kWh	551.6 ms	2.2%
RCS-30	38.8 kWh	511.3 ms	0.9%

The mean values of obtained results are also displayed in Table 2. HS saves more energy than NPA because it dynamically turns hosts into low-power mode, however, since resources are limited and without brownout, HS also experiences higher average response time and SLA violation ratio. Assuming the average response time and SLA violation ratio of QoS constraints should be below 1 second and 1% respectively, we can conclude that the brownout-based algorithms, LUCF, MNCF and RCS, can save more energy than NPA and HS while ensuring QoS by reducing average response time and SLA violation ratio. The reason lies in that the brownout-based algorithms reduce energy by deactivating a set of containers and improve the QoS compared with the overloaded situation. And the performance differences of brownout-based algorithms are due to the different selections of deactivated containers. For the comparison of brownout-based algorithms, when more optional utilization percentage is provided, the algorithms perform better. Therefore, in practice, our software system works better if more containers are allowed to be deactivated, which also means that better performance of brownout-based approach can be achieved when more containers are configured as optional.

5.7. Scalability discussion

The design and implementation of BrownoutCon are based on Docker Swarm, therefore, the performance of BrownoutCon is relevant to the scalability of Docker Swarm. Scalability tests on Docker

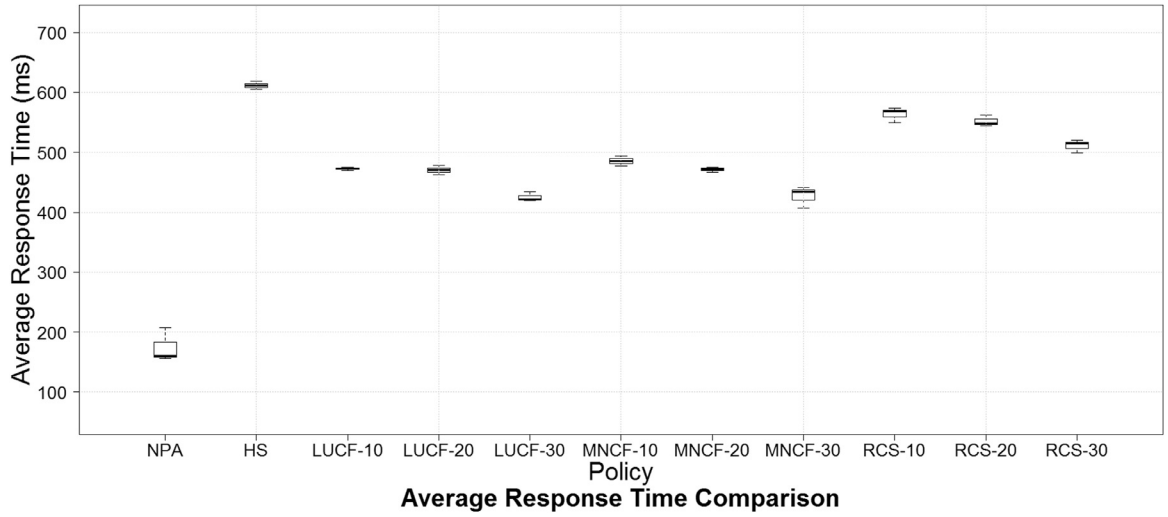

Fig. 9. Average response time comparison.

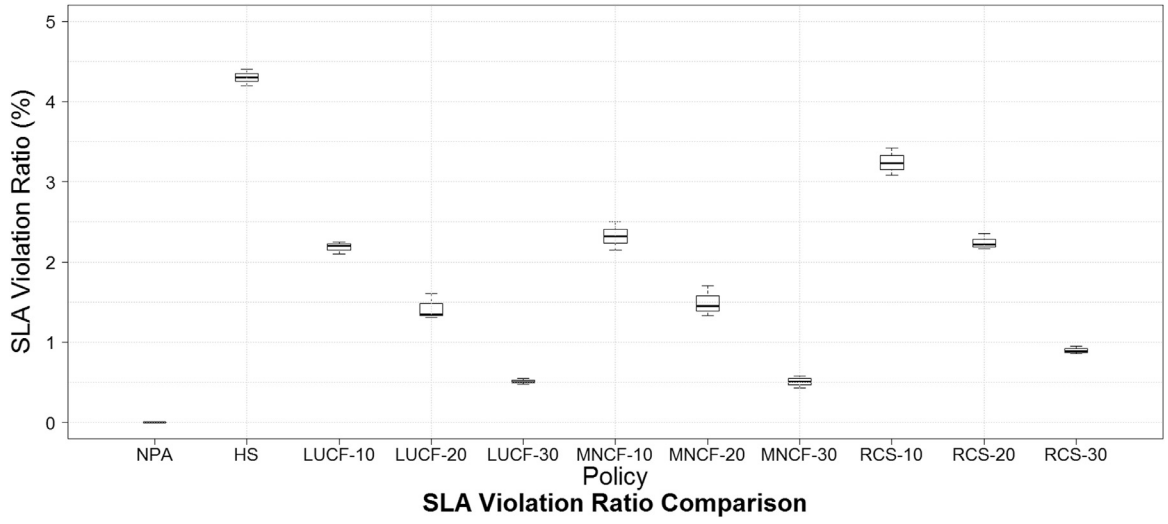

Fig. 10. SLA Violation ratio comparison.

Swarm have been conducted in Luzzardi (2015) with 1000 nodes and 30,000 containers, which shows that Docker Swarm is highly scalable system.

6. Conclusions and future work

In this paper, we proposed the design and development of a software system based on brownout and containers for energy-efficient clouds, called BrownoutCon. BrownoutCon is transparent system based on Docker Swarm for containers management and does not require to modify the default configurations of Docker Swarm via using its APIs. BrownoutCon can be customized for implementing brownout-based algorithms, which dynamically activates or deactivates containers to handle overloads and reduce energy consumption. The experiments conducted on Grid'5000 infrastructure show that the brownout-based algorithms in BrownoutCon are able to reduce energy consumption while ensuring QoS. The proposed software can be applied in the container-based environment as well as future research in brownout area.

As for future work, we would like to enable BrownoutCon to be available in other container environments, such as Kubernetes and Apache Mesos. We propose to extend the prediction algorithm to handle network congestion. We would also like to investigate memory-intensive workloads to enable BrownoutCon to be applied to more generic workloads. In addition, we plan to develop algorithms for integrated management of all resources of cloud data centers including cooling systems Gill and Buyya (2018) to significantly reduce their energy consumption; and implement them in BrownoutCon software system.

Software availability: We released BrownoutCon as open source, and it can be downloaded from: https://github.com/Cloudslab/BrownoutCon.

Acknowledgments

This work is partially supported by China Scholarship Council (CSC) and Australia Research Council (ARC) Discovery Project. We thank Marcos Assuncao from INRIA (France) for providing the access to Grid'5000 infrastructure. We thank Editor-in-Chief (Prof. Paris Avgeriou and Prof. David C. Shepherd), Area Editor (Prof. Helen Karatza), and anonymous reviewers for their excellent comments on improving the paper. We also thank Sukhpal Singh Gill and Shashikant Ilager for their comments on improving this paper.

References

Adhikary, T., Das, A.K., Razzaque, M.A., Alrubaian, M., Hassan, M.M., Alamri, A., 2017. Quality of service aware cloud resource provisioning for social multimedia services and applications. Multimed. Tools Appl. 76 (12), 14485–14509.

Arroba, P., Moya, J.M., Ayala, J.L., Buyya, R., 2015. Dvfs-aware consolidation for energy-efficient clouds. In: Proceedings of the International Conference on Parallel Architecture and Compilation. IEEE, pp. 494–495.

Baresi, L., Guinea, S., Quattrocchi, G., Tamburri, D.A., 2016. Microcloud: a container-based solution for efficient resource management in the cloud. In: Proceeding of the IEEE International Conference on Smart Cloud. IEEE, pp. 218–223.

Bawden, T., 2016. Global warming: data centres to consume three times as much energy in next decade, experts warn. http://www.independent.co.uk/environment/global-warming-data-centres-to-consume-three-times-as-much-energy-in-next-decade-experts-warn-a6830086.html.

Belog1azov, A., Buyya, R., 2012. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in cloud data centers. Concurr. Comput. Pract. Exper. 24 (13), 1397–1420.

Beloglazov, A., Abawajy, J., Buyya, R., 2012. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Generat. Comput. Syst. 28 (5), 755–768.

Beloglazov, A., Buyya, R., 2015. Openstack neat: a framework for dynamic and energy-efficient consolidation of virtual machines in openstack clouds. Concurr. Comput. Pract. Exper. 27 (5), 1310–1333.

Bernstein, D., 2014. Containers and cloud: from Lxc to docker to kubernetes. IEEE Cloud Comput. 1 (3), 81–84.

Boutin, E., Ekanayake, J., Lin, W., Shi, B., Zhou, J., Qian, Z., Wu, M., Zhou, L., 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In: Proceedings of the 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), pp. 285–300.

Buyya, R., Srirama, S.N., Casale, G., Calheiros, R., Simmhan, Y., Varghese, B., Gelenbe, E., Javadi, B., Vaquero, L.M., Netto, M.A.S., Toosi, A.N., Rodriguez, M.A., Llorente, I.M., Vimercati, S.D.C.D., Samarati, P., Milojicic, D., Varela, C., Bahsoon, R., Assuncao, M.D.D., Rana, O., Zhou, W., Jin, H., Gentzsch, W., Zomaya, A.Y., Shen, H., 2018. A manifesto for future generation cloud computing: research directions for the next decade. ACM Comput. Surv. 51 (5), 105:1–105:38.

Chen, Q., Chen, J., Zheng, B., Cui, J., Qian, Y., 2015. Utilization-based VM consolidation scheme for power efficiency in cloud data centers. In: Proceedings of the IEEE International Conference on Communication Workshop, pp. 1928–1933.

Delforge, P., 2014. Data center efficiency assessment - scaling up energy efficiency across the data center industry: evaluating key drivers and barriers. https://www.nrdc.org/sites/default/files/data-center-efficiency-assessment-IP.pdf

Dhiman, G., Marchetti, G., Rosing, T., 2009. vgreen: a system for energy efficient computing in virtualized environments. In: Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design. ACM, pp. 243–248.

Docker, 2017. Docker documentation | Docker documentation. https://docs.docker.com/

Dou, W., Xu, X., Meng, S., Zhang, X., Hu, C., Yu, S., Yang, J., 2016. An energy-aware virtual machine scheduling method for service QOS enhancement in clouds over big data. Concurr. Comput. Pract. Exper. 29 (14), 1–20.

Fan, M., Han, Q., Yang, X., 2017. Energy minimization for on-line real-time scheduling with reliability awareness. J. Syst. Softw. 127, 168–176.

Gill, S.S., Buyya, R., 2018. A taxonomy and future directions for sustainable cloud computing: 360 ° view. ACM Comput. Surv. 51 (5), 104:1–104:33. doi:10.1145/3241038.

Goiri, Í., Katsak, W., Le, K., Nguyen, T.D., Bianchini, R., 2013. Parasol and greenswitch: Managing datacenters powered by renewable energy. In: Proceedings of the ACM SIGARCH Computer Architecture News, 41. ACM, pp. 51–64.

Grid'5000, 2017. Grid'5000:home. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home

Han, Z., Tan, H., Chen, G., Wang, R., Chen, Y., Lau, F.C.M., 2016. Dynamic virtual machine management via approximate Markov decision process. In: Proceedings of the 35th Annual IEEE International Conference on Computer Communications, pp. 1–9.

Hightower, K., Burns, B., Beda, J., 2017. Kubernetes: Up and Running: Dive Into the Future of Infrastructure. "O'Reilly Media, Inc.".

Kaur, T., Chana, I., 2015. Energy efficiency techniques in cloud computing: a survey and taxonomy. ACM Comput. Surv. (CSUR) 48 (2), 22.

Kim, K.H., Beloglazov, A., Buyya, R., 2011. Power-aware provisioning of virtual machines for real-time cloud services. Concurr. Comput. Pract. Exper. 23 (13), 1491–1505.

Klein, C., Maggio, M., Årzén, K.-E., Hernández-Rodriguez, F., 2014. Brownout: building more robust cloud applications. In: Proceedings of the 36th International Conference on Software Engineering, pp. 700–711.

Kong, F., Liu, X., 2015. A survey on green-energy-aware power management for datacenters. ACM Comput. Surv. (CSUR) 47 (2), 30.

Kozhirbayev, Z., Sinnott, R.O., 2017. A performance comparison of container-based technologies for the cloud. Future Generat. Comput. Syst. 68, 175–182.

Lavallée, B., 2014. Data center energy: Reducing your carbon footprint | data center knowledge. http://www.datacenterknowledge.com/archives/2014/12/17/undertaking-challenge-reduce-data-center-carbon-footprint

Li, Z., Yan, C., Yu, X., Yu, N., 2017. Bayesian network-based virtual machines consolidation method. Future Generat. Comput. Syst. 69, 75–87.

Liu, K., Aida, K., Yokoyama, S., Masatani, Y., 2016. Flexible container-based computing platform on cloud for scientific Workflows. In: Proceedings of the International Conference on Cloud Computing Research and Innovations. IEEE, pp. 56–63.

Liu, Z., Chen, Y., Bash, C., Wierman, A., Gmach, D., Wang, Z., Marwah, M., Hyser, C., 2012. Renewable and cooling aware workload management for sustainable data centers. In: Proceedings of the ACM SIGMETRICS Performance Evaluation Review, 40. ACM, pp. 175–186.

Luzzardi, A., 2015. Scale testing docker swarm to 30,000 containers - Docker blog. https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/

Mastelic, T., Oleksiak, A., Claussen, H., Brandic, I., Pierson, J.-M., Vasilakos, A.V., 2015. Cloud computing: survey on energy efficiency. ACM Comput. Surv. (CSUR) 47 (2), 33.

Naik, N., 2016. Building a virtual system of systems using docker swarm in multiple clouds. In: Proceedings of the IEEE International Symposium on Systems Engineering (ISSE). IEEE, pp. 1–3.

Newman, S., 2015. Building microservices. "O'Reilly Media, Inc.".

Pahl, C., Brogi, A., Soldani, J., Jamshidi, P., 2017. Cloud container technologies: a state-of-the-art review. IEEE Trans. Cloud Comput. 1–14.

Rodriguez, M.A., Buyya, R., 2018. Container-based cluster orchestration systems: a taxonomy and future directions. Softw. Pract. Exper. 1–22.

Santos, E.A., McLean, C., Solinas, C., Hindle, A., 2018. How does docker affect energy consumption? evaluating workloads in and out of Docker containers. J. Syst. Softw. 146, 14–25.

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

Teng, F., Yu, L., Li, T., Deng, D., Magoulès, F., 2016. Energy efficiency of VM consolidation in IAAS clouds. J. Supercomput. 1–28.

Toosi, A.N., Qu, C., de Assunção, M.D., Buyya, R., 2017. Renewable-aware geographical load balancing of web applications for sustainable data centers. J. Netw. Comput. Appl. 83, 155–168.

Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al., 2013. Apache Hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing. ACM, p. 5.

Xu, M., Buyya, R., 2017. Energy efficient scheduling of application components via brownout and approximate Markov decision process. In: Proceedings of the 15th International Conference on Service-Oriented Computing, pp. 206–220.

Xu, M., Buyya, R., 2019. Brownout approach for adaptive management of resources and applications in cloud computing systems: a taxonomy and future directions. ACM Comput. Surv. 52 (1), 8:1–8:27. doi:10.1145/3234151.

Xu, M., Dastjerdi, A.V., Buyya, R., 2016. Energy efficient scheduling of cloud application components with brownout. IEEE Trans. Sustainable Comput. 1 (2), 40–53.

Xu, M., Nadjaran Toosi, A., Buyya, R., 2018. Ibrownout: an integrated approach for managing energy and brownout in container-based clouds. IEEE Trans. Sustainable Comput. 1–14.

Zhang, X., Wu, T., Chen, M., Wei, T., Zhou, J., Hu, S., Buyya, R., 2019. Energy-aware virtual machine allocation for cloud with resource reservation. J. Syst. Softw. 147, 147–161.

Minxian Xu received the B.Sc degree in 2012 and the MSc degree in 2015, both in software engineering from University of Electronic Science and Technology of China. He is working towards the PhD degree at the Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, the University of Melbourne, Australia. His research interests include resource scheduling and optimization in cloud computing with special focus on energy efficiency. He has co-authored several peer-reviewed papers published in prominent international journals and conferences, such as ACM Computing Surveys, IEEE Transactions on Sustainable Computing, IEEE Transactions on Automation Science and Engineering, Concurrency and Computation: Practice and Experience, International Conference on Service-Oriented Computing.

Rajkumar Buyya is a Redmond Barry Distinguished Professor and Director of the Cloud Computing and Distributed Systems (CLOUDS) Laboratory at the University of Melbourne, Australia. He is also serving as the founding CEO of Manjrasoft, a spin-off company of the University, commercializing its innovations in Cloud Computing. He served as a Future Fellow of the Australian Research Council during 2012–2016. He has authored over 625 publications and seven text books including "Mastering Cloud Computing" published by McGraw Hill, China Machine Press, and Morgan Kaufmann for Indian, Chinese and international markets respectively. He is one of the highly cited authors in computer science and software engineering worldwide (h-index=123, g-index=271, 79,800+ citations). Dr. Buyya is recognized as a "Web of Science Highly Cited Researcher" in 2016, 2017 and 2018 by Thomson Reuters, a Fellow of IEEE, and Scopus Researcher of the Year 2017 with Excellence in Innovative Research Award by Elsevier for his outstanding contributions to Cloud computing. He served as the founding Editor-in-Chief of the IEEE Transactions on Cloud Computing. He is currently serving as Co-Editor-in-Chief of Journal of Software: Practice and Experience, which was established over 45 years ago. For further information on Dr. Buyya, please visit his cyberhome: www.buyya.com