Journal of Network and Computer Applications 119 (2018) 97–109

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

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

Application deployment using Microservice and Docker containers: Framework and optimization

Xili Wan[a], Xinjie Guan[a,*], Tianjing Wang[a], Guangwei Bai[a], Baek-Yong Choi[b]

[a] *Department of Computer Science and Technology, Nanjing Tech University, Nanjing, 211816, China*
[b] *School of Computer Science and Electric Engineering, University of Missouri-Kansas City, 5110 Rockhill Rd, MO, 64110, USA*

ARTICLE INFO

Keywords:
Application deployment
Microservice architecture
Docker container

ABSTRACT

To improve the scalability and elasticity of application deployment and operation in cloud computing environments, new architectures and techniques are developed and studied, e.g., microservice architecture, and Docker container. Especially, Docker container enables the sharing on operation system and supporting libraries, which is more lightweight, prompt and scalable than Hypervisor based virtualization. These features make it ideally suit for applications deployed in microservice architecture. However, existing models and schemes, which are mostly designed for Hypervisor based virtualization techniques, fall short to be efficiently used for Docker container based application deployment. To take the benefits of microservice architecture and Docker containers, we explore the optimization of application deployment in cloud data centers using microservice and Docker containers. Our goal is to minimize the application deployment cost as well as the operation cost while preserving service delay requirements for applications.

In this paper, we first formulate the application deployment problem by examining the features of Docker, the requirements of microservice-based applications, and available resources in cloud data centers. We further propose a communication efficient framework and a suboptimal algorithm to determine the container placement and task assignment. The proposed algorithm works in a distributed and incremental manner, which makes it scalable to massive physical resources and diverse applications under the framework. We validate the efficiency of our solution through comparisons with three existing strategies in Docker Swarm using real traces from Google Cluster. The evaluation results show that the proposed framework and algorithm provide more flexibility and save more cost than existing strategies.

1. Introduction

Cloud computing is experiencing a rapid proliferation. To take its benefits, e.g., scalable, elastic, agile, cost efficient, more applications are migrated from private infrastructures to cloud data centers. By properly allocating the physical resources in clouds, e.g., CPU, memory, storage, and network resources, to various cloud applications, computing resources are efficiently utilized and the cost is reduced for application deployment and operation. Nevertheless, cost efficiency for application deployment in clouds is one of the biggest concerns for service providers and cloud operators. To achieve this goal, resource allocation and management have been widely studied for applications deployed under different architectures or paradigms.

To support elastic computing in a cloud with multi-tier structure, resource can be scaled by adjusting the amount of physical resources assigned to each deployed application (Liu et al., 2015; Wei et al., 2010). There are two ways for resource scaling. One is to add more **virtual machines (VMs)** (Jiang et al., 2013) which is referred as the *horizontal scaling*. The other way is to allocate more resources to deployed VMs (Shi et al., 2016), which is referred as the *vertical scaling*. However, the horizontal scaling may take dozens of seconds to deploy VMs or wake up a **physical machine (PM)**. For the vertical scaling, the supporting techniques, for example the Dynamic Voltage Frequency Scaling (DVFS), need additional supports from both the host operation system (OS) and guest OS, which will inevitably incur additional latency (Baccarelli et al., 2015; Shojafar et al., 2016).

Various architecture, paradigms are studied to accelerate the procedures and optimize the cost of applications' development and deployment over clouds. The emergent container based virtualization, Docker in particular, becomes an alternative for deploying applications

* Corresponding author.
E-mail addresses: xiliwan@njtech.edu.cn (X. Wan), xjguan@njtech.edu.cn (X. Guan), wangtianjing@njtech.edu.cn (T. Wang), bai@njtech.edu.cn (G. Bai), choiby@umkc.edu (B.-Y. Choi).

https://doi.org/10.1016/j.jnca.2018.07.003
Received 18 December 2017; Received in revised form 10 May 2018; Accepted 6 July 2018
Available online 09 July 2018

in clouds. Containers share not only physical resources but also the operating system as well as supporting libraries, while traditional Hypervisor based VMs only offer an abstraction at the hardware level. Meanwhile, microservice architecture (Martin and James) decouples complicated applications into lightweight and loosely linked components. Each component independently performs a microservice, and could be replaced or updated without the involvement of other components. The advent of container and microservice architecture provide the possibility to improve the scalability and elasticity of application development. Nevertheless, due to the inherent differences, existing studies and mechanisms on resource allocation for Hypervisor based VM and monolithic architecture cannot be directly applied for resource allocation using Docker containers and microservice-based applications.

Some work has been done towards utilizing containers to deploy microservice-based applications in practice. Zhou et al. (2018) studied the microservice scheduling problem aiming to maximum the revenue of deploying microservices. Guerrero et al. (2018) designed genetic approaches to determine the amount of resources assigned to each microservice, and how to efficiently scale while the workload changes. Fazio et al. (2016) summarized the difficulties in scheduling and resource management in deploying microservices with dynamic user requests and heterogeneous settings in clouds. However, most existing work did not consider the features of containers, e.g., dynamic resource scaling at fine granularity, image layering and libraries reuse, which would impact the resource usage efficiency and application deployment cost.

We noticed the importance of considering the feature of Docker containers, and developed a preliminary approach to dynamically allocate resources for applications in data centers using docker container (Guan et al., 2017). Nevertheless (Guan et al., 2017), focused on the feature of Docker containers, but assumed that applications are deployed in monolithic architecture so that containers for a single application are homogeneous. The limitations of monolithic architecture, such as inefficiency in updating and shipping, drive us to explore resource allocation for microservice-based application deployment.

We explore both the opportunities and challenges in deploying cloud applications using Docker containers under microservice architecture. Specifically, we consider the following unique features for application deployment with Docker containers under microservice architecture:

- Each application may consist of heterogeneous microservices, each of which requires different amount of resources and supporting libraries.
- The number and capacity of containers are adaptively determined based on not only applications' requirements but also available resources on PMs.
- The containers deployment cost is related to available supporting libraries on PMs and required libraries of applications.
- Resource management and application execution functions are decoupled, and resource management is performed in a distributed manner.

Considering the above-mentioned features, we target to optimize the deployment of microservice-based applications with Docker containers by minimizing the application deployment cost as well as operation cost. Through solving the application deployment problem, we aim to answer the following questions: 1) Where to place the container-based microservices for each application so that the deployment and operation cost could be minimized? 2) How much amount of resources, including computational resources and network resources, would be assigned to each container without conflicting with the resource constraints while satisfying users' requirements? In addition, to be applicable in large scale data centers with fluctuated amount of workloads, the application deployment solution is expected to be scalable and adaptive to changes.

In order to tackle the application deployment problem in practice, we first propose a novel framework named ADMD, in which the resource assigned to each application could be adaptively adjusted by the microservice controller for the application in a distributed manner. Then we develop a cost-efficient and scalable algorithm for each microservice controller to determine the deployment of execution containers and task assignment. The main contribution of this paper are summarized below:

- We mathematically model the microservice-based application deployment problem to minimize total cost, taking the features of Docker container into consideration.
- A scalable framework is presented for the problem which could dynamically adjust the amount of resources allocated to each application based on its requirements and status.
- Concerning for the difficulties of the application deployment problem, we decompose the origin problem as small-sized sub-problems, which could be independently solved in the proposed framework in a distributed manner.

The rest of this paper is organized as follows. Section 2 briefly introduces container-based virtualization and microservice architecture. Section 3 briefly overviews related technologies including related studies specifically within the context of resource allocation in cloud data centers. We formulate the cost for application deployment using microservice applications and Docker containers as an optimization problem in Section 4. Due to the hardness of this optimization problem, we decompose it into subproblems, and solve them with a scalable framework and a communication-efficient algorithm. Section 5 demonstrates the microservice-based application deployment framework, and Section 6 details the design about the resource allocation algorithms. Evaluations are shown in Section 7, and the conclude remarks are summarized in Section 8.

2. Background

This section provides a brief background on the container-based virtualization and microservice architecture including definition and motivation in using these techniques for application deployment in data centers.

2.1. Container-based virtualization

As an alternative of hypervisor based virtualization, container based virtualization has been proposed to meet the need for saving system resources (Soltesz et al., 2007), and attracts many attentions in recent years. Compared with traditional Hypervisor-based virtualization that provides isolation at hardware abstraction layer, container-based virtualization offers operation system level abstraction and isolation. It allows system resources and operation system to be shared among multiple containers.

Google Cloud Platform and Microsoft Azure has started to support container based virtualization in their cloud services. Due to its light weight size, prompt deployment and migration (Felter et al., 2015; Dłaz et al., 2016), container could further improve the efficiency and flexibility of resource allocation in cloud data centers (Felter et al., 2015; Xavier et al., 2013). compared the capacity of container based virtualization with traditional Hypervisor based VMs, and validated that container outperforms VM in performance, lightweight and scalability.

There are different implementations of container, e.g., Solaris 10, Linux Vserver, and Docker. Among these variations of container, Docker, which is an open source project, further enables the reuse of common supporting libraries. In addition, from Docker v1.11.1, the amount of resources assigned to a running container could be dynamically adjusted on the fly.

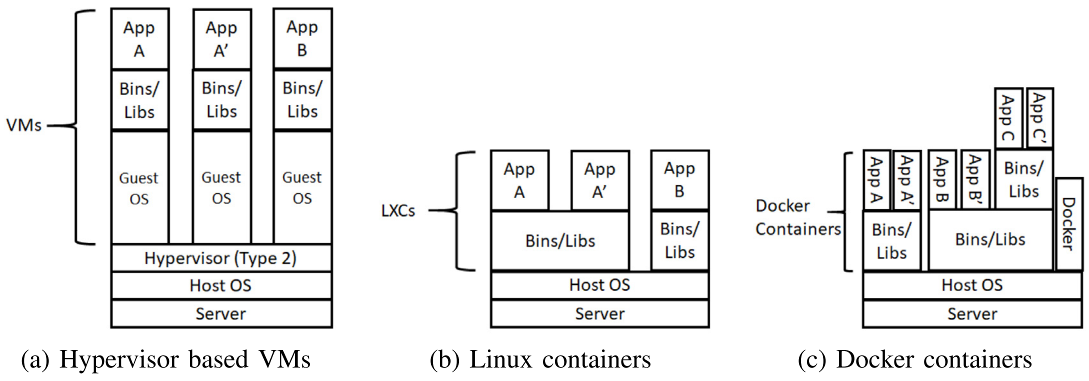

Fig. 1. Comparison for Hypervisor based VMs, Linux containers (LXCs) and Docker containers.

Fig. 1 illustrates the differences among Hypervisor based VMs, Linux containers (LXC) (Soltesz et al., 2007), and Docker container. As shown in Fig. 1(a), each Hypervisor based VM (Type 2) needs to run a separate operating system and install all supporting libraries before deploying applications, while the kernel system could be shared by Linux containers as shown in Fig. 1(b). Docker further enables image layering that makes it possible to share the supporting libraries as shown in Fig. 1(c). This key feature makes containers, especially the Docker containers, much more lightweight, prompt and scalable than Hyperisor based VMs.

2.2. Microservice architecture

Till now the most common architecture to deploy applications is monolithic. In the “monolithic” architecture, each deployment unit, i.e., a container or a VM, is an autonomous entity that handles all the functions. For example, a container for the HTTP server-side application would process everything for HTTP requests, including performing domain logic, executing database relate procedures, generating and sending HTML views. Despite that applications in the monolithic architecture are easy to deploy, it is difficult to update and less scalable. Any small modification would induce updating and re-deploying the entire system (Namiot and Sneps-Sneppe, 2014), which inevitably delay application's release cycle.

To overcome this limitation of “monolithic” architecture, microservice architecture (Martin and James) is proposed that decouples complicated applications into lightweight and loosely coupled components. Each component independently performs a microservice with separate source code repository, and could be updated without the involving of the others. In addition, considering different resource requirements of different application components, microservice architecture supports independently scaling up of each component. Thus, microservice architecture provides the possibility to improve the scalability and elasticity to application development.

Currently, microservice architecture is still in its initial phase with many challenges before being widely employed in application deployments. Container-based virtualization, due to its light-weight and prompt, is treated as a good match to accelerating the application of microservices architecture in practice.

3. Related work

As an essential issue in cloud computing, application deployments have been widely studied for various applications. For the applications that are deployed in a multi-tier structure, such as Hadoop MapReduce (Dean and Ghemawat, 2008), a single computing job could be partitioned into multiple tasks by a central manager. Then, these tasks are assigned to a set of worker nodes placed on servers with limited physical resources. In order to fully utilize the limited resources while ensuring the quality of services, work (Alicherry and Lakshman, 2013; Xu and Tang, 2014) discussed the strategies to determine the number and placement of worker nodes. In addition, after placing a proper number of worker nodes, certain amount of tasks are assigned and executed on those worker nodes. The assignment of tasks and the scheduling of different tasks on worker nodes impact the performance of applications (Kwon et al., 2012; Le et al., 2014; Kc and Anyanwu, 2010; Sandholm and Lai, 2010). Kwon et al. (2012) and Le et al. (2014) suggested dividing tasks with a large amount of computing requirement into pieces so that all the tasks could be completed within a similar time. Kc and Anyanwu (2010) and Sandholm and Lai (2010) proposed tasks scheduling strategies to ensure the fairness among different jobs. It is worth to note that in the multi-tier structure, resources could be easily scaled and adapted to various application workloads by adjusting the number of worker nodes assigned to each task (Liu et al., 2015; Wei et al., 2010).

Although the multi-tier structure supports resource allocation at fine granularity, it is mainly used for big data computation rather than complicated applications, e.g. game hosting, video conference. For those complicated applications, specific execution environments and functions are required, including operation systems, and various supporting libraries. These environments can hardly be built by using multi-tier structure.

To satisfy the specific requirements of these complicated applications, VMs are employed to simulate the environments of physical servers while allowing resource sharing. Multiple VMs could be consolidated or isolated from each other in the same PM. The mapping from VMs to PMs, named VM placement problem have been studied towards different objectives, such as maximizing the number of embedded VMs (Tang et al., 2007; Wang et al., 2011), minimizing the cost and maximizing the revenue (Ardagna et al., 2012), minimizing the energy consumption (Aroca et al., 2016; Xiao et al., 2013; Beloglazov et al., 2012; Guan et al., 2014), and improving the reliability (Machida et al., 2010).

In addition, resource management for applications with varied workload has been taken into consideration as well. Jiang et al. (2013) studied automatic scaling from horizontal direction, while (Shi et al., 2016; Lama et al., 2013) focused on adaptive scaling from vertical direction. However, in horizontal scaling, deploying or waking up a physical server may take up to dozens of seconds, while vertical scaling needs additional supports from both host operation system and guest operation system. Therefore, dynamic scaling is not quite efficient and practical for resource allocation using Hypervisor based VM.

As an alternative of Hypervisor based VM, container is light weight and could be promptly deployed. Motivated by the benefits of containers, effort has been done towards deploying application using containers. Various objectives have been considered while satisfying resource constraints. Sureshkumar and Rajesh (2017) aimed to minimize the energy consumption in container based resource allocation through dynamically switches the status of containers between sleep and awake based on a predefined threshold. Ahmed et al. (2017)

Table 1
Notations used.

Notation	Explanation
G^p	The physical network
N^p	The set of physical nodes
$c^p(i)$	Residual resources on PM i, $i \in N^p$
$\mathbb{K}$	The set of supporting libraries
$s^p(i)$	An indicator that if physical node i is in sleep or awake status
Λ	The set of applications needed to be deployed
G^a	The application G^a, $G^a \in \Lambda$
N^a	The set of microservice N^a for application G^a
$w^a(u)$	Total number of requests for application G^a that need microservice u
$t^a(u)$	The amount of unit time to finish an instance of microservice u with a unit amount of physical resources
$T^a(u)$	The maximum allowed service delay to complete an instance of microservice u
$F^a(u)$	Expected amount of internal traffic between controller and containers for microservice u in application G^a
$C^a(u)$	The minimum amount of computational resources demanded for guarantee the service delay bound
$I(a)$	The PM embedding the controller for application G^a
$U^{node}(i, a)$	The node cost for installing a microserive of application G^a on physical node i
P^s	The baseline cost of a PM
$P^l(k)$	The installation cost for setting up Library k, $k \in \mathbb{K}$
P^o	The operation cost of performing a unit amount of workload
$U^{link}(i, a)$	The link cost for communication between the controller and the Docker container on PM i for application G^a
$D(i, j)$	The distance between PM i and PM j
$f(i, a, u)$	The amount of traffic passing through the path between PM i for microservice u and the controller for application G^a
$x(i, a, u)$	A variable indicating number of requests for microservice u' that is embedded in physical node i
$s(i)$	Binary variable if any micro-service is embedded in physical node i or not
$l^p(k, i)$	An indicator that if supporting Library k, $k \in \mathbb{K}$ available on physical node i
$l^a(k, u)$	An indicator that if supporting Library k, $k \in \mathbb{K}$ is required for microservice u in application G^a
$l(k, i)$	Binary variable that library k would be install on PM i

designed a process state synchronization mechanism to assist the migration of computation instance between different clouds with the minimized network interruption. In (Ahmed et al., 2015), they also summarized recent proposed approaches for deployment of delay sensitive applications in mobile cloud computing, with the emphasizing on the ways to reduce application response time and user interaction delay. Zhang et al. (2017) formulated the container placement problem as ILP to minimize the deployment cost including host energy cost and image transmission cost. Considering the hardness of the application deployment problem, artificial intelligent based algorithms have been employed to determine the placement of containers, e.g., ant colony optimization based algorithm in (Kaewkasi and Chuenmuneewong, 2017), artificial fish swarm based algorithm in (Li et al., 2016). Besides, evolutionary computation algorithm (Vigliotti and Batista, 2014) and game theory (Xu et al., 2014) were applied to solved the container placement problem as well (Vigliotti and Batista, 2014). utilized containers to retrieve resource usage status and develops Knapsack based algorithm and Evolutionary Computation algorithm to place containers (Xu et al., 2014). partitioned a job into a set of sub tasks and uses game theory to solve the resource allocation problem at container level (Tao et al., 2017). designed a fuzzy inference system node selection algorithm for container deployment.

The above mentioned work introduced various mechanisms and approaches for adopting container in application deployment. However, most of them model the container placement problem as a knapsack problem, without taking the features of Docker container into consideration. Specifically, to make fully use of Docker containers' feature, there are two main differences compared with traditional knapsack problem:

- Docker container supports operation system and libraries reusing, so that the total deployment cost of a set of containers on the same PM, is not the sum of each of their deployment cost.
- The number of containers and their capacities could be dynamically adjusted on the fly based on time varying applications' workloads and available physical resources.

Facing the differences in Docker container based application deployment, we proposed an initial solution in (Guan et al., 2017), where Application Oriented Docker Container (AODC) based resource allocation is developed. In AODC, resource management and application execution functions are decoupled and performed on different kinds of containers, and resource management is performed in a distributed manner. Despite presenting the possibility of improving cost efficiency and scalability in (Guan et al., 2017) by adapting Docker container than using traditional Hypervisor-based VMs, this work is based on monolithic architecture in which the containers for an application are assumed to be homogeneous and autonomous.

To deploy complex applications, researchers partitioned the applications into subtasks. For each subtask, a container based replica would be deployed towards certain optimization goal. In (Singh and Peddoju, 2017), a container-based framework is designed to deploy applications in microservice architecture with minimum downtime for application integration and delivery. Zhou et al. (2018) aimed to maximize the revenue by properly scheduling jobs with and without deadline while considering the dependence relationship between subtasks (Guerrero et al., 2018). illustrated genetic approaches to determine the amount of resources assigned to each microservice, and studied how to efficiently scale while the workload changes. These studies discussed the tasks assignment and resource allocation for applications in microservice architecture. Even though containers are utilized as the basic units to execute microservices, most of these studies focused on the partition among multiple subtasks, but omitted the features of Docker containers.

Different from existing work and our preliminary study (Guan et al., 2017), we utilize the unique features of Docker containers and microservice architecture for 1) improving resource usage efficiency, 2) reducing the deployment cost, 3) building an accurate and practical model of application deployment. In addition, to deal with a large amount of applications, we design a framework for deploying applications using microservice and Docker in a distributed manner. With this framework, the resources assigned to each application would be monitored and dynamically adjusted independently by its controller. Therefore, the proposed solution is scalable while applications' workloads or the number of applications change.

4. Problem formulation

In this section, we model the applications deployment application using microservice and Docker as an optimization problem, aiming to minimize the deploying cost while satisfying the application's QoS requirements. Here, we define the total deployment costs of an application as the summation of operation cost for each individual microservice in the application. The notations used in this section is listed as in Table 1.

4.1. System model

We consider a physical network $G^p(N^p)$ with a set of PMs. Each PM is equipped with limited physical resources. Here, we use computational resources as an example to model the resource allocation problem. We assume that these PMs are identical in capacity and price. However, the PMs may install different libraries in advance to support different microservices for applications.

A set of applications are deployed with resources in the physical network. Each application can be represented as a set of loosely-coupled microservice. Each microservice u of an application takes responsibility to complete a specific subfunction for requests to this application. To provide fine-grained elasticity, each microservice is deployed and scaled independently without impacting the rest of the application.

A microservice u of application G^a is further considered as a tuple $(w^a(u), t^a(u), T^a(u), F^a(u), l^a(k, u))$. Here, $w^a(u)$ denotes the expected amount of requests for microservice u. Each request for microservice u could be completed by a unit amount of computational resource in $t^a(u)$ unit amount of time. When more computational resources are assigned for serving this request, it could be finished within a less time. $T^a(u)$ specifies the maximum allowed service delay to complete a request for microservice u of application G^a. $F^a(u)$ specifies the expected total amount of internal traffic between the controller and all Docker containers for microservice u of application G^a. $l^a(k, u)$ is a binary indicator, that equals to 1, if supporting library k is required for deploying microservice u.

To coordinate the communications among microservices, each application has a controller to collects intermediate results from microservice and forwards them to the next microservices requiring the data. Assume that all the Docker containers serving microservice u are started and terminated at the same time, and at least $C^a(u) = \frac{w^a(u) * t^a(u)}{T^a(u)}$ unit amount of computational resources are demanded to guarantee the service delay bound $T^a(u)$.

4.2. Cost-minimized application deployment using Microservice and Docker

Given a physical network and a set of applications, each application consists of different microservices. We want to determine the number and placement of Docker containers, and the amount of requests assigned to each Docker container so that the total cost of applications is minimized. The cost of an application is defined as the summation of the cost for each individual microservice of this application. Fundamentally, the cost of each microservice of application G^a comes from the node cost U^{node} and link cost U^{link}. Specifically, U^{node} includes deployment cost and execution cost of Docker containers, while U^{link} quantifies the communication cost between Docker containers.

We adapt the widely accepted linear power model (Krishnan et al., 2011; Chen et al., 2015; Guan et al., 2015) to estimate the cost of a PM i that consists of baseline cost and operation cost. Here, we assume that the operation cost is proportional to the workload assigned to the PM i. To save deployment cost, we utilize PMs that support the sleep/awake mode, and assume no baseline cost for a PM in sleep mode. Besides the baseline cost and operation cost, deploying an Docker container may bring additional cost for installing and configuring necessary supporting libraries of the application.

Therefore, for the node cost, three factors are taken into consideration when counting, i.e., the cost for waking up an inactive PM, the cost for installing supporting libraries, and the cost for serving assigned requests. Thus, the node cost $U^{node}(i, a)$ for installing microservices of application G^a on a PM i, is modeled as shown in Eq. (1).

$$U^{node}(i, a) = P^s \cdot s(i) + \sum_k P^l(k) \cdot l(k, i) + \sum_{u \in N^a} P^o \cdot \frac{x(i, a, u) \cdot C^a(u)}{w^a(u)} \tag{1}$$

P^s is the cost for wake-up a PM. The binary variable $s(i)$ is set to 1, if a PM would be switched from off status to on, otherwise 0. $P^l(k)$ is the cost for installing library k on a PM, and $l(k, i)$ is a binary variable that library k would be install on PM i. Variable $x(i, a, u)$ denotes the number of requests assigned to the Docker container on PM i for microservice u. Since a request is not allowed to be decomposed, $x(i, a, u)$ is a non negative integer. $C^a(u)$ is the amount of physical resources required to complete all requests for microservice u of application G^a before the deadline. $w^a(u)$ is the total amount of requests for microservice u. $\frac{x(i,a,u) \cdot C^a(u)}{w^a(u)}$ is the amount of physical resources needed to complete all assigned requests for microservice u on PM i before the deadline. Currently, the voltage and frequency of PMs are assumed to be fixed in our model. In the future, we would consider dynamic voltage and frequency for further cost saving as in (Baccarelli et al., 2015).

The link cost U^{link} describes the data exchange cost between the controller and Docker containers for a single application, which is highly related to the considered technology, e.g., virtual links using TCP/IP connections (Cordeschi et al., 2012). Here, we consider the general scenario without assumptions on the underlying technology as in (Chowdhury et al., 2009). We further assume that traffic between the controller and Docker containers is unsplitable, and always goes through the shortest path for simplicity. Then, the link cost is modeled as the product of the path length and expected traffic amount between the controller and containers. When the controller of an application G^a has been determined, the link cost for application G^a between PM i, and the selected PM to place controller $I(a)$, $U^{link}(i, a)$, is modeled as:

$$U^{link}(i, a) = \sum_{u \in N^a} D(I(a), i) \cdot f(i, a, u) \tag{2}$$

$D(I(a), i)$ is the length of the shortest path between PM $I(a)$ and PM i. Note $D(I(a), i)$ is known when PM $I(a)$ is determined as the request of deploying an application G^a arrives in a data center. This process would be further discussed in Section 5. We do not consider link failure and reconfiguration, so $D(I(a), i)$ is fixed for application G^a and PM i, $i \in N^p$. $f(i, a, u)$ indicates the amount of traffic passing through this path for microservice u of application G^a. Assume that $f(i, a, u)$ is proportional to the amount of requests assigned to the Docker container on PM i. Thus, we have $f(i, a, u) = \frac{F^a(u) \cdot x(i, a, u)}{w^a(u)}$. To save communication cost, Docker containers with heavier traffic are preferred to be embedded near the controller.

As we consider the node cost as well as the communication cost, a coefficient $\alpha \in [0, 1]$ is employed to balance the influence of the node cost and the communication cost to satisfy various application and performance requirements. Then, by summing up the balanced cost for deploying a set of applications, the objective function of applications deployment using microservice with Docker could be formulated as follows:

$$\min \left\{ \sum_{i \in N^p} \sum_{G^a \in \Lambda} (\alpha \cdot U^{node}(i, a) + (1 - \alpha) \cdot U^{link}(i, a)) \right\} \tag{3}$$

In addition, this optimization problem is subject to the following constraints:

- The available physical resource is limited on each PM. We restrict that the total provisioned workloads would not exceed the maximum available physical resources on each PM.

$$\sum_{G^a \in \Lambda} \sum_{u \in N^a} \frac{x(i, a, u) \cdot C^a(u)}{w^a(u)} < c^p(i), \forall\ i \in N^p \tag{4}$$

- Similar as existing work, it is guaranteed that each request for microservice u would be assigned to exactly one Docker container.

$$\sum_{i \in N^p} x(i, a, u) = w^a(u), \forall\ u \in N^a, G^a \in \Lambda \tag{5}$$

- Before assigning workload to a PM i, the status of PM i should be checked. If it is in sleep mode, it will be switched to awake status.

$$\sum_{G^a \in \Lambda} \frac{x(i, a, u)}{w^a(u)} - s(i) \leq s^p(i), \forall\ i \in N^p, u \in N^a \tag{6}$$

- Concerning on the possible differences between required supporting libraries and the installed libraries on PM i, all additional libraries should be installed to make sure a mircoservice u could be executed on a PM i.

$$\sum_{G^a \in \Lambda} \frac{l^a(k, u) \cdot x(i, a, u)}{w^a(u)} - l(k, i) \le l^p(k, i), \ \forall\, i \in N^p, u \in N^a \quad (7)$$

· The variable $s(i)$ that indicates if a PM i should be switched from sleep to awake status, and the variable $l(k, i)$ that indicates if a library k would be installed on PM i, are binary variables. Constraint is included to confine their range.

$$s(i), l(k, i) \in \{0, 1\}, \forall\, i \in N^p \quad (8)$$

· The variable $x(i, a, u)$ denotes the number of requests for microservice u of application G^a that are assigned to PM i. We assume that the workload for a single request cannot be further decomposed. The variable $x(i, a, u)$ is an integer variable.

$$x(i, a, u) \in \{0, 1, \ldots\}, \forall\, i \in N^p, G^a \in \Lambda, u \in N^a \quad (9)$$

As above presented, our aim is to optimize the weighted total cost of PMs and communications under a set of constraints by solving the optimization problem Eq. (3) subject to Eqs. (4)–(9). We refer this optimization problem for *Application Deployment using Microservice and Docker container* as the *ADMD* problem. Note that this ADMD problem is NP-hard, since the well known bin packing problem can be reduced to the special case of this problem, when no common supporting libraries can be shared by any two mircoservices.

5. A framework for deploying microservice-based applications with docker

Considering the hardness of the ADMD problem, and scalability requirements of deploying cloud applications, we develop a framework for deploying microservice-based applications with Docker, as well as a resource allocation algorithm that runs independently on each controller in a distributed manner. This algorithm acquires the resources' status from a limited number of Docker Engines and makes decisions for the Docker container's placement and task assignment for microservices of an application.

Before presenting the resource allocation algorithm, we illustrate the framework for applications deployment using microservice and Docker (ADMD framework), which is the base of the algorithm described in Section 6. In the ADMD framework, application requests are processed as microservices on **Execution Containers (ECs)**. The allocation and management of resources for applications are decentralized and performed on **Microservice Controller (MSCs)**. Unlike the Hypervisor based VM placement, the number of ECs and their demands on physical resources are dynamically determined by not only the applications' workload but also the available resources in the data center. The overview of the ADMD framework is illustrated in Fig. 2.

As presented in Fig. 2, when deploying an application, resources are allocated to the application as a set of containers distributed on multiple PMs. In particular, each application has a MSC and at least one EC. A **MSC** makes resource allocation decisions, requests resources for ECs, tracks the task status on ECs, and manages the life cycle of ECs. **ECs** complete the assigned tasks, and report to the MSC about their task execution status compared with the expected progress. In addition, the MSC collects and distributes the intermediate data generated by microservices for the application. Based on the execution status of ECs, the MSC would adjust allocated resources to ECs, balance ECs' workloads, or migrate ECs. If the execution status of an EC is behind schedule, the MSC could 1) dynamically add more resources for this EC, 2) balance its workload to other ECs, or 3) migrate this EC to another PM with enough resources.

Both MSCs and ECs are embedded on PMs. Fig. 3 depicts the main components in a PM. Here, each PM has a host OS, on which Docker runs an engine to maintain the operating environment for containers, assist embedding containers, and isolate containers running on the same PM. In addition, we introduce a **scheduler** on each PM to manage the life cycle of the MSCs and ECs running on this PM. When receiving an application deployment request, the gateway (as shown in Fig. 2) distributes the request to 1 p.m. according to predetermined policy, e.g., load balancing, location preference. The scheduler on the selected PM, e.g., *physical machine 1* in the Fig. 3, creates a MSC and assigns resources to it based on the application's requirements and available resources on the PM. It may forward the application deployment request to other schedulers, if there is not enough physical resource on the selected PM. For the other schedulers on the PMs that have not been selected to place MSCs, e.g., *physical machine 2* in the Fig. 3, they 1) report local available resources when receiving queries from a MSC, 2) approve or reject the MSCs' requirements on creating new ECs, and 3) recycle the physical resources when ECs complete the assigned tasks. In addition, the Docker Engine enables the OS kernel and common supporting libraries to be shared by multiple ECs, so that aggregating applications that share libraries could further save costs by reducing redundancy. As demonstrated in Fig. 3, by placing the EC for application A (EC1-A) and the EC for application B (EC1-B) on the same *physical machine 2*, the common libraries (cycled in the Fig. 3) could be shared rather than be installed separately for each application.

We illustrate the workflow of deploying an application in a data center in Fig. 4. When an application request arrives, the gateway dispatches the request to a PM based on certain policy, e.g., load balancing, location preference (steps (1)). The scheduler on the selected PM (*PM 1 in* Fig. 4) then initiates a MSC and assigns physical resources for the MSC. It then replies with a message indicating the MSC has been successfully created (step (2)). The MSC queries a set of PMs about their available resources, supporting libraries installed (step (3)), and receives replies from the PMs (step (4)). Based on the application's requirements, and available resources on PMs, the MSC makes resource allocation decisions using the proposed EPTA algorithm (Algorithm 1, discussed in Section 6), and requests resources on selected PMs (*PM 2 and PM 3 in* Fig. 4) for creating ECs based on the decision (step (5)). If the request has been approved by the scheduler on the selected PM (*PM 2* in Fig. 4), the PM would provision resources, create an EC for the microservice, and send a confirmation message to the MSC (steps (6)). After that, tasks are assigned to the EC according to the resource allocation decision. PM may reject EC creating requirement (step (7)) when it does not have enough resource. In this case, the MSC has to repeat the step (5) and request for creating the ECs on other PMs (step (8) and (9)). A MSC may repeat the steps (4–6) to create multiple ECs based on the resource allocation decision. When all ECs are created, the application has been successfully deployed (step (10)).

The created ECs work on assigned tasks and update task execution status to the MSC. Based on the real time workload of the application, a MSC may dynamically adjust the number, location and assigned resources of ECs. When the application is deactivated, the MSC is terminated and its resources are collected by the scheduler.

Under this framework, physical resources allocated for each application could dynamically shrink or expand based on the application's requirements, real time workload and available resources in the data center. Since the resource allocation decision is made by each MSC independently, the framework is scalable in data centers with a large amount of physical resources and diversity applications. To further minimize the cost of applications deployment in the framework, we develop a scalable algorithm to solve this problem in Section 6.

6. Algorithms

Considering the hardness of the ADMD problem and scalability requirements of deploying cloud applications, we develop a scalable algorithm that runs independently on each MSC in a distributed manner to acquire the resources' status from a limited number of Docker Engines and to make resource allocation decisions.

To reduce the computational complexity of origin ADMD problem, we decompose it to sub-problems ADMD-s. Every ADMD-s aims to

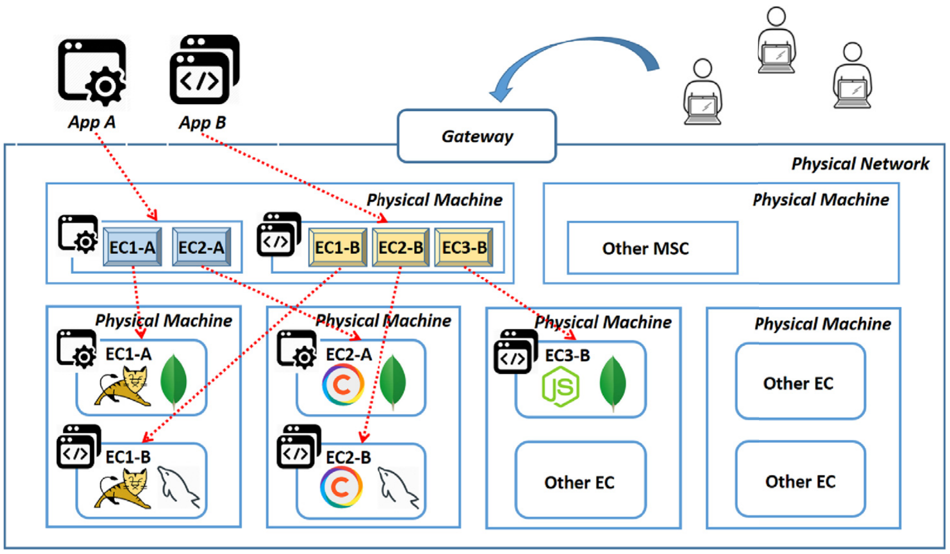

Fig. 2. Framework for deploying microservice-based applications with Docker.

minimize the deployment cost for a single application G^a (Eq. (10)) subjected to the constraints on available resources, total workloads of microservices, and existence of supporting libraries (Eqs. (11)–(16)).

$$\min\left\{\sum_{i\in N^p}(\alpha\cdot U^{node}(i)+(1-\alpha)\cdot U^{link}(i))\right\} \tag{10}$$

subject to:

$$\sum_{u\in N^a}\frac{x(i,u)\cdot C^a(u)}{w^a(u)} < c^p(i),\ \forall\ i\in N^p \tag{11}$$

$$\sum_{i\in N^p}x(i,u)=w^a(u),\ \forall\ u\in N^a \tag{12}$$

$$\frac{x(i,u)}{w^a(u)}-s(i)\le s^p(i),\ \forall\ i\in N^p,\ u\in N^a \tag{13}$$

$$\frac{l^a(k,u)\cdot x(i,u)}{w^a(u)}-l(k,i)\le l^p(k,i),\ \forall\ i\in N^p,\ u\in N^a \tag{14}$$

$$s(i),\ l(k,i)\in\{0,1\},\ \forall\ i\in N^p \tag{15}$$

$$x(i,u)\in\{0,1,\ldots\},\ \forall\ i\in N^p,\ u\in N^a \tag{16}$$

By solving the sub-problem ADMD-s, a MSC only need to determine for a single application where to place ECs and the amount of requests assigned to each EC. Compared to origin ADMD problem, the solution space for ADMD-s has been significantly reduced. In addition, the computation is decentralized to MSCs that are allocated on different PMs, so that the workload of the gateway server is alleviated. We name the EC Placement and Task Assignment algorithm for microservice-based application as EPTA, and present it in Algorithm 1.

Algorithm 1
EC Placement and Task assignment Algorithm for microservice-based application i (EPTA).

```
Input: Physical network in region(s) $G_r^p(N_r^p)$; Resource allocation
  request $G^A(N^A)$; A set of neighbor region IDs $set_n$
Output: Allocation decision $X=\{x(i,u)\mid i\in N_r^p\}$ for microservice $u$
  1: Query PMs in the local region, e.g., a rack, for available
     resources
```

(*continued on next page*)

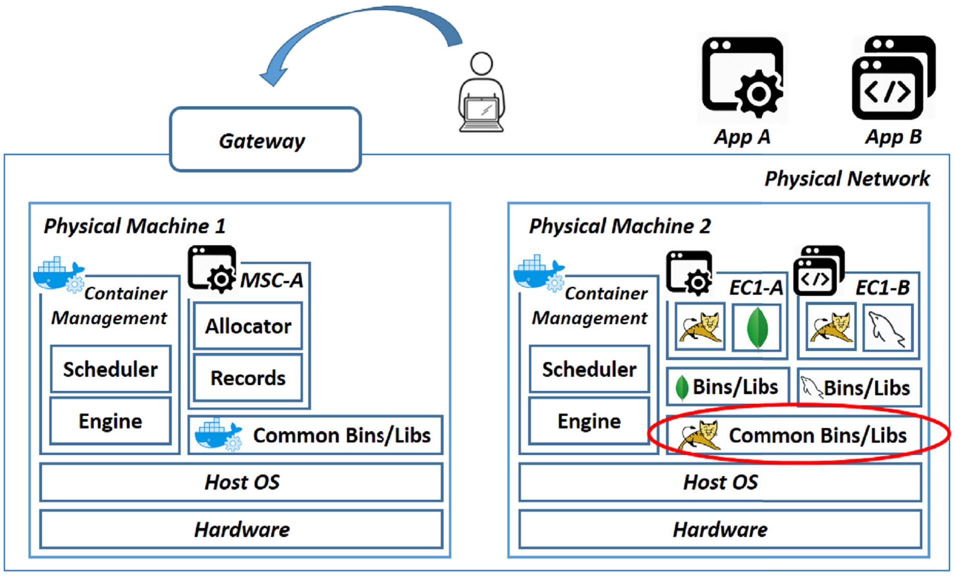

Fig. 3. Components in a physical machine: *physical machine 1* embeds microservice controllers, while *physical machine 2* embeds execution containers.

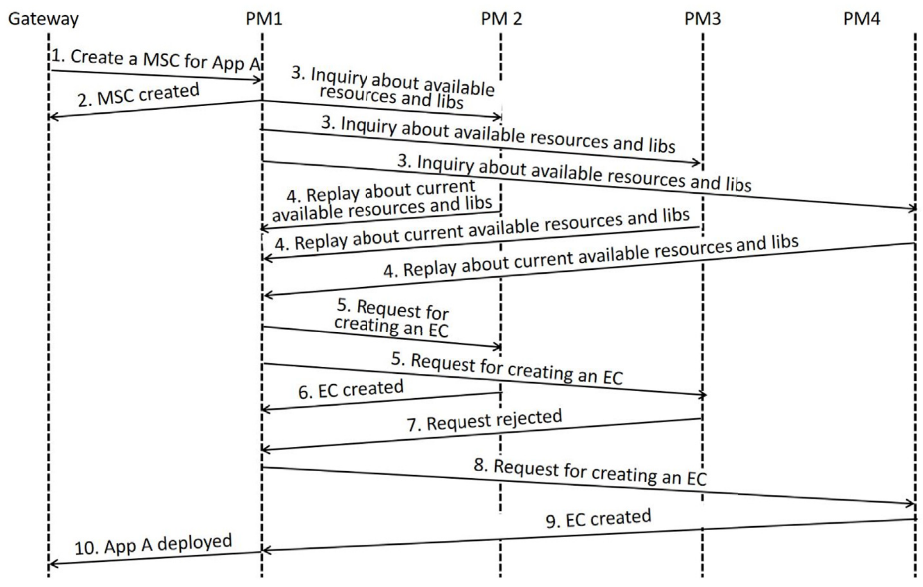
Gateway
PM1
PM 2
PM3
PM4
1. Create a MSC for App A
2. MSC created
3. Inquiry about available resources and libs
3. Inquiry about available resources and libs
3. Inquiry about available resources and libs
4. Replay about current available resources and libs
4. Replay about current available resources and libs
4. Replay about current available resources and libs
5. Request for creating an EC
5. Request for creating an EC
6. EC created
7. Request rejected
8. Request for creating an EC
9. EC created
10. App A deployed

Fig. 4. An example of the communication sequence of deploying an application in a data center based on ADMD framework.

Algorithm 1 (*continued*)

```
2: Solve the optimization problem (10) subject to (11)–(16) using
   LP solver, e.g., CPLEX
3: If a feasible solution exist, send resource request to PM $i$ for the
   amount of $\frac{x(i,u)\cdot C^A(u)}{w^A(u)}$ for each variable $x(i, u)$ in the solution
4: Wait for the response from PM $i$
5: If the EC creation request has been approved by PM $i$, record $x(i, u)$ into $directory_u$, and update the status of PM $i$
6: If the EC creation request has been rejected by PM $i$, record $x(i, u)$ into $set_u$, and update the status of PM $i$
7: When the responses for all requests have been received, but $set_u$
   is not null
8: Update the resource allocation request by deducting allocated
   microserives
9: Extend searching area, e.g., a pod, and call the EPTA algorithm
   again
```

As shown in Algorithm 1, the MSC starts querying available physical resources within a small local region, e.g., a rack (Step (1)). Based on the available resources reported by the Docker Engine on each PM, the ADMD-s problem (Step (2)) could be solved by adopting Linear Programming (LP) solver, e.g., CPLEX. Then, the MSC requests physical resources based on the optimization problem's solution (Steps (3)), and waits for the responses from PMs (Step (4)).

If the request is approved, an EC is built on the selected PM, and the status of the PM, e.g., available resources, installed supporting libraries, is updated (Step (5)). The MSC records the deployed EC $x(i, u)$ into $directory_u$, so that it could track each EC for dynamic adjustments.

If the request is rejected, this PM is marked as infeasible and the assigned task is moved to set_u (Step (6)). When there is no feasible solution for the optimization problem, or some tasks have not been successfully assigned, the MSC queries PMs in a larger scale, e.g., a pod, and solves the optimization problem again for not assigned tasks (Steps (7–9)).

Algorithm 2
Microservice embedding procedures on ECs.

```
1: When receive a resource query from a MSC, share current status
2: When an EC creation request has been received, check if its
   resource demands could be satisfied with residual resources
3: If there is not enough resources, send a response
4: If there are enough resources, reserve the resources, create an EC
   and send a response
5: For each embedded EC, build a channel between the EC and its
   MSC for control communications
```

Corresponding to the EPTA algorithm running on MSC for searching available resources and deploying cloud applications, PM responses the queries and requests from MSC as described in Algorithm 2. When a PM receives a resource inquiry, the scheduler on this PM shares the current status of this PM including residual resources and supporting libraries installed with the MSC (Step (1)). When the scheduler receives an EC creation request, it checks if residual resources could satisfy the demands of the EC (Step (2)). If current available resource cannot meet demands, the scheduler replies a response to reject the request along with updated status (Step (3)). Otherwise, the scheduler reserves demanded resources, installs necessary supporting libraries, and downloads the EC image from repository. After that, an EC has been successfully created and the scheduler sends a confirmation message to the MSC (Step (4)). In addition, for each embedded EC, the scheduler on the PM maintains a secure channel for control communications and intermediate results updates (Step (5)).

Note that in the EPTA algorithm, a MSC initially only queries a small area rather than the entire physical network, then incrementally expands the searching area when there are not enough available resources. Each PM would be queried at most once, and in most cases, only a portion of the physical network would be queried. Therefore, the EPTA algorithm is communication-efficient.

Meanwhile, the EPTA algorithm executes in a distributed manner by the MSCs. Each MSC determines the placement of ECs and job assignments for an application. Therefore, the algorithm scales with the number of applications. In addition, for each application, the searching area is incrementally increased, and only a part of the physical network for most applications would be queried. Thus, the EPTA algorithm is scalable while the size of physical network grows.

7. Evaluations

In this section, we evaluate the performance of our EPTA algorithm in different context through trace-driven simulation studies. All evaluations are based on real traces from Google Cluster Traces.

To demonstrate the necessity of considering the Docker containers' feature, we compare the EPTA algorithm with three strategies implemented in Docker Swarm Strategies, including Spread, Binpack and Random, with respect to different aspects. Docker Swarm is an orchestration framework that is currently integrated into Docker Engine for container swarm creation and management as well as application deployment. In addition, even through the performance differences between Hypervisor-based VM and container have been widely studied and compared, e.g., (Felter et al., 2015; Xavier et al., 2013), for readers' convenience, we still include a Hypervisor based VM embedding algorithm in the comparison to show the differences between container and Hypervisor-based VM in application deployment. The three strategies of Docker Swarm and Hypervisor-based VM embedding are explained as follows:

- Spread selects the PM with the least number of containers to place new containers.
- Binpack selects the PM that is the most packed to place new containers.
- Random selects PMs randomly regardless of PMs' resource usage status.
- Optimal-VM selects PMs based on the optimal solution of Hypervisor-based VM embedding problem achieved by CPLEX

For the sake of simplification and to simulate large-scale scenarios, we implement the five algorithms and base our simulation on real traces from Google Cluster Traces. These traces specify more than 1,000,000 tasks along with their arrival time and resource requirements during 7 h. Each task maps a set of Linux processes. Considering the large amount of tasks in these traces, only the tasks arrived in the first hour are used in our evaluation. In addition, due to different granularity between a task and a microservice, we group every 50 tasks as a single microservice, and use the arrival time for the first task as the arrival time for the microservice request.

The physical network is randomly generated by Overview-NetworkX, and the maximum available computational resource of each PM is randomly chosen from (1,2,4,8,16) core(s). In addition, we randomly set the number of supporting libraries required by one microservice between (Liu et al., 2015; Baccarelli et al., 2015), while the supporting libraries are randomly selected from a pool of 10 libraries. The size of each libraries is set between [0.03,0.15] GB.

7.1. On the number of PMs

We first compare the *total deployment cost* of our EPTA algorithm with three strategies of Docker Swarm and the Optimal-VM while the number of available PMs varies from 50 to 150, then check the *PM active rate* in the network and the *average number of deployed libraries* on each PM. Through this set of evaluation, we want to demonstrate the performance of EPTA in physical networks with different scales. Here, *PM active rate* is defined as the ratio of the number of active PMs to the total number of PMs, while the *average number of deployed libraries* is the average number of supporting libraries installed on each PM for microservices deployed on this PM. The more active PMs induce more baseline cost, while a larger average number of deployed libraries indicates more library installation cost. For all the five algorithms, they are executed online, with no knowledge about next arriving applications. Thus, they make decisions based only on current status of the physical network and the application. We also take a look at the *computation time* of each algorithm.

As shown in Fig. 5, EPTA outperforms other deployment algorithms

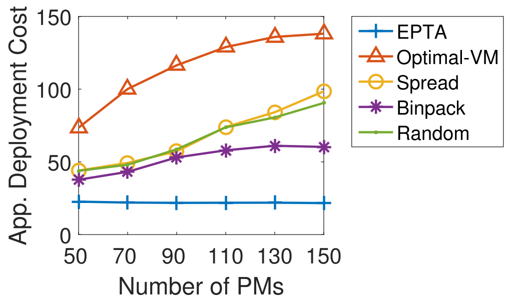

Fig. 5. Average application deployment cost with varied number of PMs.

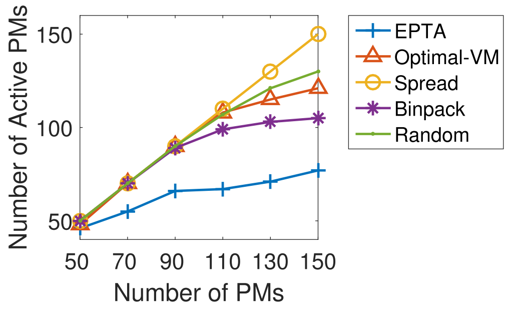

Fig. 6. Number of active PMs with varied number of PMs.

in total deployment cost, while Optimal-VM spends the most deployment cost. The application deployment costs of the three Docker swarm strategies are between that of EPTA and Optimal-VM. As the number of PMs increases from 50 to 150, the total deployment costs of EPTA slightly drop. This is because when more PMs are available, the potential to find a better PM to place a microservice increases. However, the total deployment costs of other strategies and algorithm increase, since they occupy more PMs as shown in Fig. 6.

In Fig. 6, for load balancing, Spread always utilize all the PMs to deploy the coming applications, while EPTA, Optimal and Binpack use a portion of the PMs when there is enough number of PMs. Random also utilizes a large portion of PMs, thus, its deployment cost grows as the number of PMs in the network grows as well.

We further look into the average number of supporting libraries on each PM in Fig. 7. Based on Fig. 7, EPTA has the smallest number of libraries installed on each PM. This validates that EPTA eliminates redundancy by sharing supporting libraries and operation systems. Compared with EPTA, all the other strategies and algorithm do not consider libraries reuse when deploying the applications. Thus, more supporting libraries are installed on PMs. Furthermore, as the number of PMs in the network rises, containers are distributed on more PMs, which reduces the average number of libraries on each PM for EPTA and the three Docker swarm strategies. Note that, for Optimal-VM,

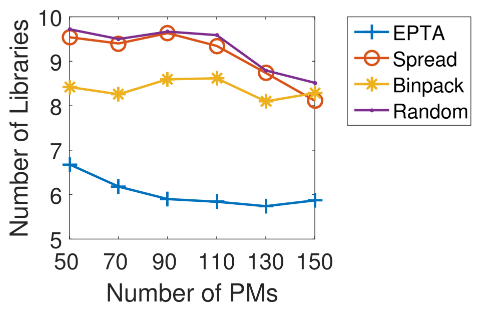

Fig. 7. Average number of supporting libraries on each PM with varied number of PMs.

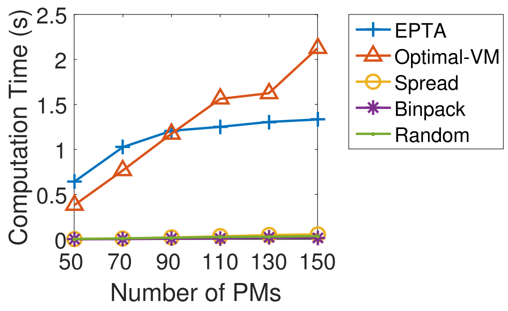

Fig. 8. Average computation time with varied number of PMs.

every embedded VM includes an OS and all supporting libraries. The average number of libraries on each PM when using Optimal-VM is proportional to the number of VM on each PM and the number of libraries in this VM. Therefore, we did not include the number of libraries of Optimal-VM in Fig. 7.

In addition, the computation time for the five strategies and algorithms are compared in Fig. 8. As demonstrated in Fig. 8, the three Docker Swarm strategies are a bit faster than EPTA and Optimal-VM for the cases of 50, 70 and 90 PMs. This is because that both Optimal-VM and EPTA include linear problems and use LP solver. However, it is worth to note that the time complexity of EPTA does not grow exponentially as the network scales, since EPTA incrementally extends the searching area rather than using the entire physical network as an input.

7.2. On the number of microservices

We examine the performance of EPTA for deploying applications with single or multiple microservices in a mid-sized physical network. Specifically, we vary the number of microservices in each application from 1 to 5 for the physical network with 90 PMs, and compare the total deployment costs, number of active PMs, average number of supporting libraries, and computation time of the five application deployment strategies and algorithms. Here, to exclude the impact of applications' workload and number of supporting libraries, the total workload of an application and the total number of supporting libraries used in an application keep the same. In other words, when there are multiple microservices in an application, these microservices are evenly assigned a portion of the application's total workload, and the union of their supporting libraries equals to the supporting libraries required by the application. It is worth to note that when an application only has one microservice, it could be considered as in monolithic architecture.

The application deployment costs of the five strategies and algorithms are presented in Fig. 9. It can be observed in Fig. 9 that EPTA has the smallest application deployment cost, while that of Optimal-VM increases significantly. Among the three Docker Swarm strategies, Binpack consumes the least deployment cost, while Spread takes the most. This is mainly because that Binpack trends to put microservices from the same application on the same PM, while Spread and Random trend to distribute the microservices in the network for load balancing. Therefore, Binpack saves the communication costs among microservices. In addition, as illustrated in Fig. 10 and Fig. 11, Spread and Random wake up more PMs, and install more libraries on each PM, which increase their deployment cost, compared with EPTA. Fig. 12 indicates that as the number of microservices in an application increases, the computation time of EPTA and Optimal-VM rises, because of the utilization of LP solver. In the future, we will design an efficient solver tailored for our problem to replace the common LP solver used in the EPTA algorithm.

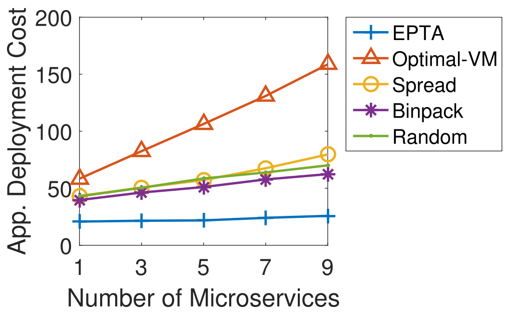

Fig. 9. Average application deployment cost with varied number of microservices.

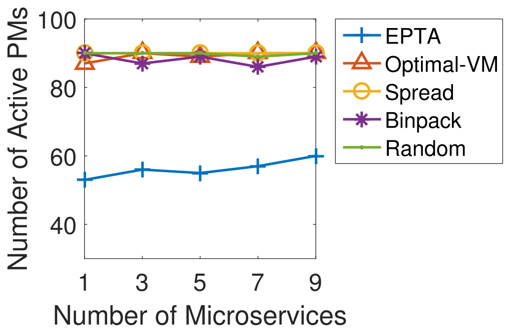

Fig. 10. Number of active PMs with varied number of microservices.

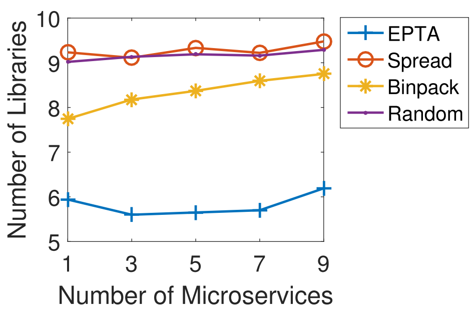

Fig. 11. Average number of supporting libraries on each PM with varied number of microservices.

7.3. On the varied size of supporting libraries

Finally, we check the impact of average size of supporting libraries on the performance of the five strategies and algorithms. The influence of the libraries sizes are examined in physical networks with different scales (90 PMs and 150 PMs).

As shown in Fig. 13 and Fig. 17, the application deployment costs of the five strategies and algorithms increases as the average library size

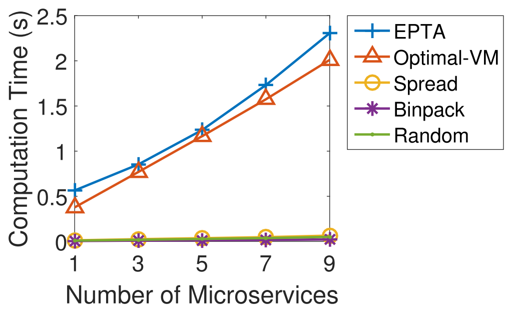

Fig. 12. Average computation time with varied number of microservices.

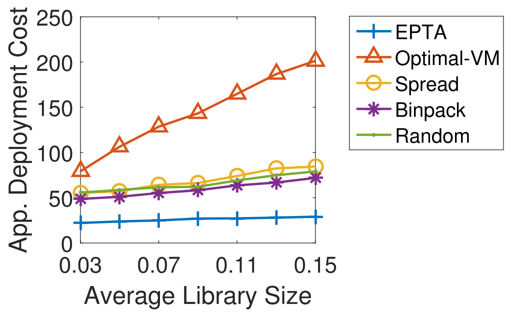

Fig. 13. Average application deployment cost with varied library size (90 PMs).

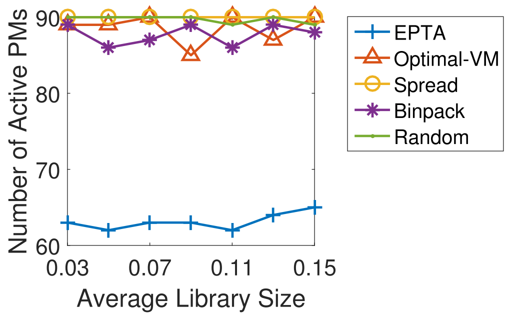

Fig. 14. Number of active PMs with varied library size (90 PMs).

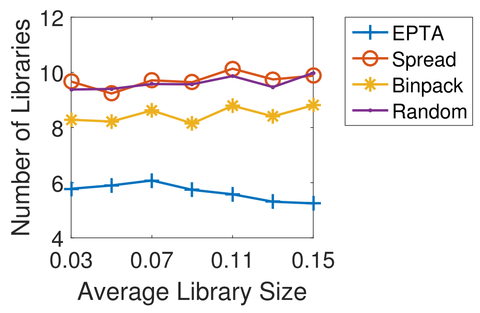

Fig. 15. Average number of supporting libraries on each PM with varied library size (90 PMs).

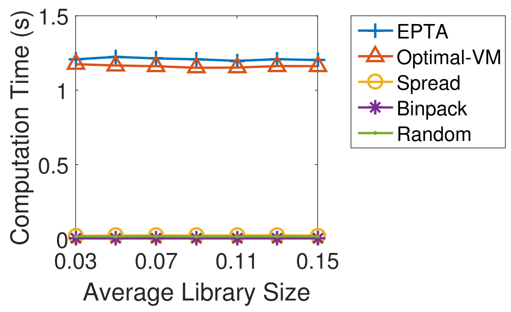

Fig. 16. Average computation time with varied library size (90 PMs).

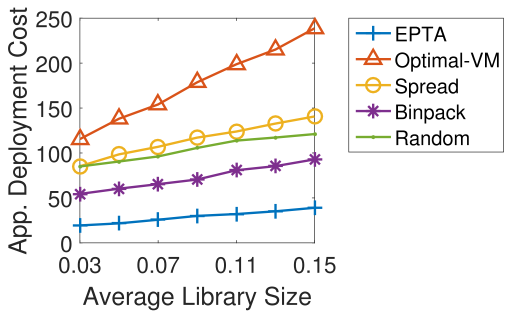

Fig. 17. Average application deployment cost with varied library size (150 PMs).

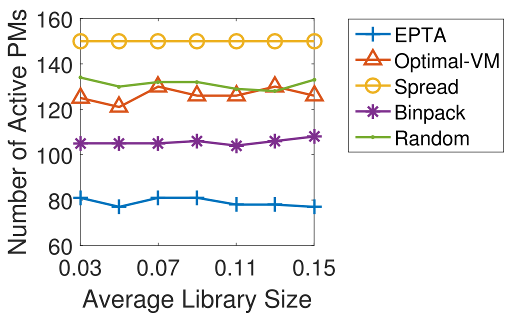

Fig. 18. Number of active PMs with varied library size (150 PMs).

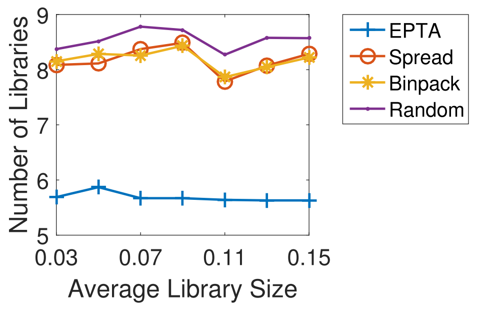

Fig. 19. Average number of supporting libraries on each PM with varied library size (150 PMs).

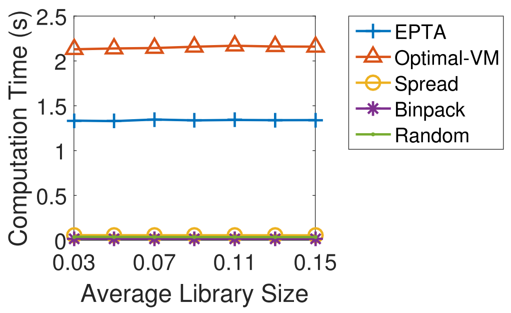

Fig. 20. Average computation time with varied library size (150 PMs).

grows. It does not lead too much changes on the number of active PMs (shown in Fig. 14 and Fig. 18) and the average number of libraries (shown in Fig. 15 and Fig. 19) for Docker Swarm strategies and Optimal-VM. However, the active PMs used by EPTA is slightly increased, and the number of libraries of EPTA drops. This indicates that EPTA could find a good balance between the baseline cost and library installation cost. Furthermore, there is no obvious change on the computation time of all the strategies and algorithms when library size varies as presented in Fig. 16 and Fig. 20.

Through the above three sets of simulations, it is observed that EPTA could improve the application deployment cost by balancing baseline cost and library installation cost. This is because 1) library reuse is fully utilized and redundancy has been alleviate; 2) the size of containers is not fixed but dynamically adopted based on application's requirements and available resources. In addition, EPTA could be efficiently scaled while the size of physical network grows. However, its computation time is still high compared to Docker Swarm strategies, since we adopt the common LP solver in our algorithm. We will reduce its time complexity in our future work.

8. Conclusion and future work

By allowing sharing on operation system as well as supporting libraries, container based virtualization offers a great opportunity for reducing application deployment cost and improving final users' experience. Considering the features and benefits of emergent Docker container, we modeled the cost efficient resource allocation for microservice-based applications with docker container, and proposed a communication-efficient and scalable resource allocation algorithm, named EPTA, to minimize the application deployment cost constrained by capacity and the service delay bound. The presence of EPTA algorithm significantly improves the deployment cost by balancing the PM waking-up cost, supporting library installation cost and communication cost. By incrementally extend searching area, it could easily scale up as the size of the cloud data center grows. We validated the efficiency of the proposed schemes with the comparison with VM placement algorithm and three strategies implemented in Docker Swarm in evaluations based on real data traces.

For future research directions, we will reduce the computation complexity in EPTA by solving the LP model in a more efficient way. In addition, we will consider other optimization goal, e.g., minimize applications' response time, energy efficiency, in our framework, and build a prototype which integrates our framework and algorithm into Docker swarm for implementation in real cloud environments.

Acknowledgement

This work is supported in part by the National Nature Science Foundation of China (Grant No. 61602235, 61501224), and the Natural Science Foundation of Jiangsu Province of China (Grant No. BK20161007).

References

Ahmed, E., Gani, A., Khan, M.K., Buyya, R., Khan, S.U., 2015. Seamless application execution in mobile cloud computing: motivation, taxonomy, and open challenges. J. Netw. Comput. Appl. 52, 154–172.

Ahmed, E., Naveed, A., Gani, A., Hamid, S.H.A., Imran, M., Guizani, M., May 2017. Process state synchronization for mobility support in mobile cloud computing. In: IEEE International Conference on Communications (ICC), pp. 1–6.

Alicherry, M., Lakshman, T., 2013. Optimizing data access latencies in cloud systems by intelligent virtual machine placement. In: IEEE INFOCOM, pp. 647–655.

Ardagna, D., Panicucci, B., Trubian, M., Zhang, L., 2012. Energy-aware autonomic resource allocation in multitier virtualized environments. IEEE Trans. Serv. Comput. 5 (1), 2–19.

Aroca, J.A., Anta, A.F., Mosteiro, M.A., Thraves, C., Wang, L., 2016. Power-efficient assignment of virtual machines to physical machines. Future Generat. Comput. Syst. 54, 82–94.

Baccarelli, E., Amendola, D., Cordeschi, N., 2015. Minimum-energy bandwidth management for qos live migration of virtual machines. Comput. Network. 93, 1–22.

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.

Chen, X., Li, C., Jiang, Y., Aug 2015. Optimization model and algorithm for energy efficient virtual node embedding. IEEE Commun. Lett. 19 (8), 1327–1330.

Chowdhury, N., Rahman, M., Boutaba, R., April 2009. Virtual network embedding with coordinated node and link mapping. In: IEEE INFOCOM, pp. 783–791.

Cordeschi, N., Patriarca, T., Baccarelli, E., 2012. Stochastic traffic engineering for real-time applications over wireless networks. J. Netw. Comput. Appl. 35 (2), 681–694.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113.

Docker. https://www.docker.com/.

Docker Swarm Strategies. https://docs.docker.com/swarm/scheduler/strategy/.

Dłaz, M., Martłn, C., Rubio, B., 2016. State-of-the-art, challenges, and open issues in the integration of internet of things and cloud computing. J. Netw. Comput. Appl. 67, 99–117.

Fazio, M., Celesti, A., Ranjan, R., Liu, C., Chen, L., Villari, M., 2016. Open issues in scheduling microservices in the cloud. IEEE Cloud Comput. 3 (5), 81–88.

Felter, W., Ferreira, A., Rajamony, R., Rubio, J., March 2015. An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172.

Google Cloud Platform. https://cloud.google.com/products/.

Google Cluster Traces. https://github.com/google/cluster-data/blob/master/TraceVersion1.md.

Guan, X., Choi, B.Y., Song, S., 2014. Topology and migration-aware energy efficient virtual network embedding for green dcs. In: 23rd International Conference on Computer Communication and Networks (ICCCN). IEEE, pp. 1–8.

Guan, X., Choi, B.Y., Song, S., 2015. Energy efficient virtual network embedding for green dcs using dc topology and future migration. Comput. Commun. 69, 50–59.

Guan, X., Wan, X., Choi, B.Y., Song, S., Zhu, J., March 2017. Application oriented dynamic resource allocation for data centers using docker containers. IEEE Commun. Lett. 21 (3), 504–507.

Guerrero, C., Lera, I., Juiz, C., 2018. Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. Springer J. Grid Comput. 16 (1), 113–135.

Jiang, J., Lu, J., Zhang, G., Long, G., 2013. Optimal cloud resource auto-scaling for web applications. In: International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 58–65.

Kaewkasi, C., Chuenmuneewong, K., Feb 2017. Improvement of container scheduling for docker using ant colony optimization. In: 9th International Conference on Knowledge and Smart Technology (KST), pp. 254–259.

Kc, K., Anyanwu, K., 2010. Scheduling hadoop jobs to meet deadlines. In: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 388–392.

Krishnan, B., Amur, H., Gavrilovska, A., Schwan, K., 2011. Vm power metering: feasibility and challenges. Perform. Eval. Rev. 38 (3), 56–60.

Kwon, Y., Balazinska, M., Howe, B., Rolia, J., 2012. Skewtune: mitigating skew in mapreduce applications. In: ACM SIGMOD International Conference on Management of Data, pp. 25–36.

Lama, P., Guo, Y., Zhou, X., 2013. Autonomic performance and power control for co-located web applications on virtualized servers. In: IEEE/ACM 21st International Symposium on Quality of Service (IWQoS), pp. 1–10.

Le, Y., Liu, J., Ergun, F., Wang, D., 2014. Online load balancing for mapreduce with skewed data input. In: IEEE INFOCOM, pp. 2004–2012.

Li, Y., Zhang, J., Zhang, W., Liu, Q., Nov 2016. Cluster resource adjustment based on an improved artificial fish swarm algorithm in mesos. In: IEEE 13th International Conference on Signal Processing (ICSP), pp. 1843–1847.

Liu, Z., Zhang, Q., Zhani, M., Boutaba, R., Liu, Y., Gong, Z., 2015. Dreams: dynamic resource allocation for mapreduce with data skew. In: International Symposium on Integrated Network Management, pp. 18–26.

Machida, F., Kawato, M., Maeno, Y., 2010. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In: IEEE Network Operations and Management Symposium (NOMS), pp. 32–39.

F. Martin and L. James. Microservice. https://martinfowler.com/articles/microservices.html.

Microsoft Azure: Cloud Computing Platform & Services. https://azure.microsoft.com/.

Namiot, D., Sneps-Sneppe, M., 2014. On micro-services architecture. Int. J. Open Inf. Technol. 2 (9), 24–27.

Overview-NetworkX. http://networkx.github.io/.

Sandholm, T., Lai, K., 2010. Dynamic proportional share scheduling in hadoop. In: Job Scheduling Strategies for Parallel Processing. Springer, pp. 110–131.

Shi, X., Dong, J., Djouadi, S., Feng, Y., Ma, X., Wang, Y., 2016. Papmsc: power-aware performance management approach for virtualized web servers via stochastic control. J. Grid Comput. 14 (1), 171–191.

Shojafar, M., Cordeschi, N., Baccarelli, E., Apr. 2016. Energy-efficient adaptive resource management for real-time vehicular cloud services. IEEE Trans. Cloud Comput. 99 1–1.

Singh, V., Peddoju, S.K., May 2017. Container-based microservice architecture for cloud applications. In: 2017 International Conference on Computing, Communication and Automation (ICCCA), pp. 847–852.

Soltesz, S., Pötzl, H., Fiuczynski, M., Bavier, A., Peterson, L., 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: ACM SIGOPS Operating System Review, vol. 41. pp. 275–287.

Sureshkumar, M., Rajesh, P., 2017. Optimizing the docker container usage based on load scheduling. In: 2nd International Conference on Computing and Communications Technologies (ICCCT), pp. 165–168.

Tang, C., Steinder, M., Spreitzer, M., Pacifici, G., 2007. A scalable application placement controller for enterprise data centers. In: Proc. 16th International Conference on World Wide Web. ACM, pp. 331–340.

Tao, Y., Wang, X., Xu, X., Chen, Y., March 2017. Dynamic resource allocation algorithm for container-based service computing. In: IEEE 13th International Symposium on Autonomous Decentralized System (ISADS), pp. 61–67.

Vigliotti, A., Batista, D., 2014. Energy-efficient virtual machines placement. In: Brazilian Symposium on Computer Networks and Distributed Systems (SBRC). IEEE, pp. 1–8.

Wang, M., Meng, X., Zhang, L., 2011. Consolidating virtual machines with dynamic bandwidth demand in data centers. In: IEEE INFOCOM, pp. 71–75.

Wei, G., Vasilakos, A., Zheng, Y., Xiong, N., 2010. A game-theoretic method of fair resource allocation for cloud computing services. J. Supercomput. 54 (2), 252–269.

Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C., 2013. Performance evaluation of container-based virtualization for high performance computing environments. In: 21st Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP). IEEE, pp. 233–240.

Xiao, Z., Song, W., Chen, Q., 2013. Dynamic resource allocation using virtual machines for cloud computing environment. IEEE Trans. Parallel Distr. Syst. 24 (6), 1107–1117.

Xu, X., Tang, M., 2014. A more efficient and effective heuristic algorithm for the mapreduce placement problem in cloud computing. In: IEEE 7th International Conference on Cloud Computing (CLOUD), pp. 264–271.

Xu, X., Yu, H., Pei, X., 2014. A novel resource scheduling approach in container based clouds. In: 17th International Conference on Computational Science and Engineering (CSE), pp. 257–264.

Zhang, D., Yan, B.H., Feng, Z., Zhang, C., Wang, Y.X., April 2017. Container oriented job scheduling using linear programming model. In: 3rd International Conference on Information Management (ICIM), pp. 174–180.
Zhou, R., Li, Z., Wu, C., Feb. 2018. Scheduling frameworks for cloud container services. IEEE/ACM Trans. Netw. 26 (1), 436–450.

Xili Wan received the BE and ME degrees from Nanjing Tech University in China. He obtained PhD degree from University of Missouri-Kansas City. He is currently an assistant professor at the school of computer science and technology, Nanjing Tech University, China. His research interests include optimization in wireless sensor networks, design and analysis of approximation algorithms, artificial intelligence and cloud computing.

Xinjie Guan received the BS degree in Computer Science from Southeast University, in 2006, and the Ph.D degree in Computer Science from University of Missouri-Kansas City, in 2015. Currently, she is an assistant professor at the School of Computer Science and Technology, Nanjing Tech University. Her research interests include network optimization, cloud computing, and mobile edge computing.

Tianjing Wang received the BS degrees in Mathematics, from Nanjing Normal University in 2000, the MS degrees in Mathematics, from Nanjing University in 2005 and the Ph.D degree in Telecommunication, from Nanjing University of Posts & Telecommunications in 2009. She worked in the post-doctoral research center of Nanjing University of Posts & Telecommunications from 2010 to 2013. Currently, she is an associate professor in the College of Science, Nanjing Tech University. Her research interests include signal processing, cognitive radio, and network optimization.

Guangwei Bai holds a B.Eng. (1983) and a M.Eng. (1986) from the Xi'an Jiaotong University in China, both in Computer Engineering, as well as a Ph.D. (1999) in Computer Science from the University of Hamburg in Germany. From 1999 to 2001, he worked at the German National Research Center for Information Technology, Germany, as a Research Scientist. In 2001, he joined the University of Calgary, Canada, as a Research Associate. Since 2005, he has been working at the Nanjing Tech University in China, as a Professor in Computer Science. His research interests are in location-based services, privacy and security, mobile crowd sensing and computing. He is a member of the ACM.

Dr. Baek-Young Choi is an Associate Professor at the University of Missouri – Kansas City (UMKC), has been in teaching and research in the broad areas of computer networks and systems including Cloud Computing, Smart Device Technologies, Internet-of-Things, Network Algorithms and Protocols, Data Storage and Management Systems, and Measurement, Analysis and Modeling of Network Traffic, Performance and Security. Prior to joining the University of Missouri – Kansas City, Dr. Choi held positions at Sprint Advanced Technology Labs, and the University of Minnesota, Duluth, as a postdoctoral researcher, and as a 3M McKnight distinguished visiting assistant professor, respectively. She published three books on network monitoring, storage systems, and cloud computing. She has been a faculty fellow of the National Aeronautics and Space Administration (NASA), U.S. Air Force Research Laboratory's Visiting Faculty Research Program (AFRL-VFRP) and Korea Telecom's - Advance Institute of Technology (KT-AIT). She is a senior member of ACM and IEEE, and a member of IEEE Women in Engineering.