







Topology-Aware Scheduling Framework for Microservice Applications in Cloud

Xin Li , Member, IEEE, Junsong Zhou, Student Member, IEEE, Xin Wei , Student Member, IEEE, Dawei Li , Member, IEEE, Zhuzhong Qian , Member, IEEE, Jie Wu , Fellow, IEEE, Xiaolin Qin , Member, IEEE, and Sanglu Lu, Member, IEEE

Abstract—Loosely coupled and highly cohesived microservices running in containers are becoming the new paradigm for application development. Compared with monolithic applications, applications built on microservices architecture can be deployed and scaled independently, which promises to simplify software development and operation. However, the dramatic increase in the scale of microservices and east-west network traffic in the data center have made the cluster management more complex. Not only does the scale of microservices cause a great deal of pressure on cluster management, but also cascading QoS violations present a substantial risk for SLOs (Service Level Objectives). In this paper, we propose a Microservice-Oriented Topology-Aware Scheduling Framework (MOTAS), which effectively utilizes the topologies of microservices and clusters to optimize the network overhead of microservice applications through a heuristic graph mapping algorithm. The proposed framework can also guarantee the cluster resource utilization. To deal with the dynamic environment of microservice, we propose a mechanism based on distributed trace analysis to detect and handle QoS violations in microservice applications. Through real-world experiments, the framework has been proved to be effective in ensuring cluster resource utilization, reducing application end-to-end latency, improving throughput, and handling QoS violations.

Index Terms—Cloud computing, microservice, quality of service, resource scheduling.

I. INTRODUCTION

CLOUD services have recently begun to make a significant shift in architecture from monolithic to hundreds or thousands of loosely coupled microservices [1]. Each microservice can be implemented, deployed, and updated independently

without affecting the integrity of the whole application. Therefore, more and more teams are adopting microservice architectures to improve the scalability, portability, maintainability, and availability of their applications [2], [3]. The microservice architecture is a design methodology for building a Web service with a suite of small services. Each service communicates with other services using lightweight mechanisms, e.g., an HTTP resource API or a remote procedure call (RPC) [4], which completely changed the assumptions that traditional cloud systems are designed with. It also means that microservices present both opportunities and challenges when it comes to improving utilization and optimizing the quality of service (QoS).

The main difference between microservices architecture-based applications and traditional monolithic applications is that monolithic applications in the data center only generate north-south traffic [5], and cloud service providers are more concerned with load balancing of user request traffic. Microservices architecture-based applications tend to generate more east-west traffic in the data center, which is caused by inter-invocations between microservices. For this scenario, there is space for further optimization in the data center. The invocation dependencies in microservice applications can be described as a tree-like topology that provides a good opportunity for optimal deployment of microservice applications [6].

However, most of the cloud data centers use straightforward schemes to make resource allocation decisions [7], [8]. They often overlook the relationship between the allocated resources and the overall performance of each individual microservice. As a result, it may cause a waste of valuable resources on some critical microservices. There are some approaches (e.g., over provisioning [9], recurrent provisioning [10]) in large-scaled cluster that tend to allocate more CPUs and memory to microservices using performance models [11], heuristic methods [12], or machine-learning [13], [14] algorithms. However, such approaches have two limitations. First, they ignore the characteristics among microservice applications, which indicate that the topological dependencies of microservices cannot be utilized. Second, they ignore the highly dynamic characteristics of microservices and clusters. These approaches not only cannot handle the dynamic Service Level Objectives (SLO) violation problems, but also their performance may be degraded when microservices change [15].

Due to the large amount of east-west traffic in microservice applications, a straightforward idea of scheduling optimization

Manuscript received 26 February 2022; revised 4 January 2023; accepted 12 January 2023. Date of publication 23 January 2023; date of current version 31 March 2023. This work was supported in part by the National Key R&D Program of China under Grant 2019YFB2102002, in part by the National Natural Science Foundation of China under Grant 61802182, and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. Recommended for acceptance by Y. Yang. (Corresponding author: Xin Li.)

Xin Li, Junsong Zhou, Xin Wei, and Xiaolin Qin are with the Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu 211106, China (e-mail: lies@nuaa.edu.cn; arithbar@nuaa.edu.cn; weixin@nuaa.edu.cn; qinxcs@nuaa.edu.cn).

Dawei Li is with the Department of Computer Science, Montclair State University, Montclair, NJ 07043 USA (e-mail: dawei.li@montclair.edu).

Zhuzhong Qian and Sanglu Lu are with the State Key Laboratory of Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, Jiangsu 210023, China (e-mail: qzz@nju.edu.cn; sanglu@nju.edu.cn).

Jie Wu is with the Center for Networked Computing, Temple University, Philadelphia, PA 19122 USA (e-mail: jiewu@temple.edu).

Digital Object Identifier 10.1109/TPDS.2023.3238751

is to consider the topology of microservice applications. That is, placing the microservices that constitute the application next to each other to reduce the communication cost between microservices and the network interference from other services. However, such kind of solution often leads to the aggregation of microservices on the hosts, and the resource managers in the data center are constrained to allocate resources rationally to improve the resource utilization.

In summary, how to effectively use the topology information embedded in microservice applications to reduce service communication overhead, improve resource utilization in deployment, and handle dynamic SLO violations is a hard problem, and we mainly aim to tackle this problem in this paper. The main contributions of this paper can be summarized as follows:

- *Topology-aware scheduling algorithm:* We analyze the characteristics of microservice and define the profile of microservice applications. And we identify the key parameters that affect the deployment of microservice applications and define the profile of cluster. Based on these, we use a deploying model to describe the deployment of microservice applications, and further propose a new algorithm (MOTAS) that aims at improving resource utilization by reducing resource fragmentation and improving application stability by reducing communication interference.
- *Scheduling framework:* To address two main dynamic problems that affect the SLO of microservice applications, we propose a topology-aware scheduling framework. In the framework, we extend the static topology-aware scheduling algorithm to the dynamic level. We propose a rescheduling mechanism to support the structural changes of microservices and minimize the impact of redeployment on applications. And we introduce a distributed tracking system to analyze the critical paths of microservice request responses. Based on the monitoring and analysis of critical paths, the automatic reconciliation of affected microservices can be achieved to guarantee the performance of microservice applications.
- *Evaluation:* We perform an evaluation in a real cluster environment to verify the effectiveness of the scheduling framework in terms of resource utilization and SLOs of service. By comparing it with the First Fit scheduling strategy and Kubernetes default scheduling strategy, we prove that our topology-aware scheduling strategy has better performances in reducing service response time, improving throughput, and increasing cluster resource utilization. Furthermore, by injecting SLOs violations manually, we prove that our scheduling framework can effectively optimize the deployment of microservices for dynamic abnormalities of clusters and ensure the quality of service of applications.

This paper is organized as follows. We introduce the background in Section II, and formulate the problem as a multi-objective optimization model in Section III. The topology-aware scheduling algorithm and framework are presented in Sections IV and V, respectively. We evaluate the performance of the proposed scheduling framework in Section VI, and discuss the

related work in Section VII. In Section VIII, we conclude the paper.

II. BACKGROUND

Microservices split monolithic applications into a large number of small microservice applications. The dramatic expansion of the number of services poses a very big challenge for service deployment and management. First, the invocations between microservice applications rely on HTTP requests or remote procedure calls, and the bottleneck of end-to-end response latency of microservice architecture applications is more serious on the network than monolithic applications. According to statistics [16], the network communication time overhead in microservices accounts for more than 30% of the overall time overhead of the application at lower loads, and can even account for more than 55% at high loads, making it necessary to optimize the microservice scheduling deployment from the network.

On the other hand, cluster users utilize cluster resources in the form of multiple users and multiple tasks in order to improve the utilization of cluster resources. The interference between workloads often leads to degradation of SLOs [17], [18], [19]. Especially in the mixed online and offline task deployment environment, a large number of offline tasks occupy bandwidth, and resource contention eventually leads to degradation of SLOs of some online services.

A typical microservice architecture usually consists of multiple layers, mainly including the access layer, application layer, and data layer. Requests are initially received by the access layer, which implements authentication, load balancing, and traffic management. And then the requests are forwarded to the application layer. The application layer consists of multiple microservices, each of which constitutes an independent functional unit, and completes the response of requests through collaborative invocations among them. The data layer includes databases and caches, which interact with some services in the application layer to complete data reading and writing operations.

Therefore, the degradation of the SLOs of a single microservice can lead to a great impact on the overall performance of microservices architecture-based applications. Preventing online service network bandwidth contention at the service deployment level is also a good way to improve service quality and user experience.

The topology graph of a microservice application can be obtained by portraying a large number of strong dependencies in the microservice application. The topology graph of a microservice application can be described as a directed graph, and the service topology graph provides room for deployment optimization. The abstract model of microservice application invocation is shown in Fig. 1 (the topology in Fig. 1 is simplified to some extent for ease of representation), where the vertices of the graph represent a microservice and the edges represent the communication relationships between microservices.

We set an associated weight for each edge of the microservice invocation relationship, which is used to mark the communication volume. The weights of the edges are normalized according to the total available bandwidth of the physical link, where zero

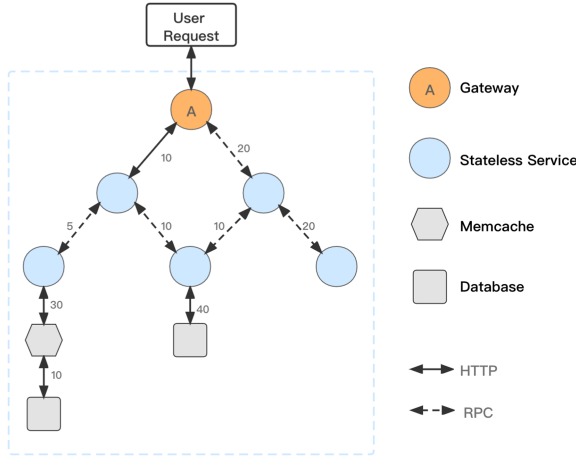


Fig. 1. Typical topology of microservice application.

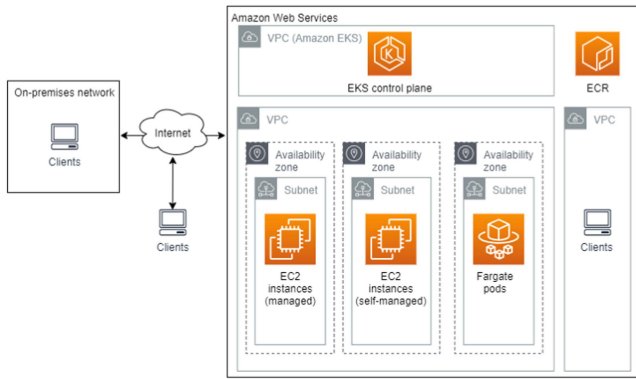


Fig. 2. Illustration of a container orchestration system (amazon EKS as a sample).

indicates no communication and a greater-than-zero weight indicates the communication level. The greater the communication level, the more susceptible the communication to interference from other applications' communication.

The topology of physical clusters is often ignored in the design of many scheduling strategies. In fact, microservices are distributed on different hosts; when a user sends a request, it may require dozens or hundreds of microservices distributed on different hosts to work together to complete the response.

At such a cluster scale, the SLOs of the application become unreliable. The reasons include: first, a large number of network calls occupy a significant portion of the time overhead [20], and the forwarding queues in the network switches amplify the performance instability [21]; second, although the replications of microservices improve the overall reliability of the application, exceptions in the individual services still can cause overall application performance to degrade [16].

Take the container orchestration service Amazon EKS [22] as an example. A typical cluster architecture designed for an application is shown in Fig. 2: Microservices are run in the form of containers in different hosts (Node); the “Node” in the EKS is composed of an Amazon EC2 instance i.e., a Virtual Machine (VM), provides the basic environment and resources for

container services. Nodes are organized into node groups, and a cluster contains multiple node groups, which are distributed on physical machines within the cluster. The physical machine can run multiple nodes (VM) through virtualization and is distributed among different racks (Racks) under the Availability Zone (AZ). These system components work together from the bottom to build the physical environment where the microservice applications run.

The network topology of the container orchestration system in Fig. 2 can abstractly describe the tree topology shown in Fig. 3, which consists of multiple layers. The apex of the tree structure indicates the availability zone (we only discuss the system architecture under a single availability zone here), and the next layer below it is the aggregation layer where the rack indicates the rack facilities of the aggregation layer. The racks are connected to the core switch through the aggregation layer switch. The next layer is the physical machine layer, using the symbol PM to indicate the set of all physical machines in the physical machine layer. The physical machines are connected to each other through the access layer switch. The next layer denotes the virtual machine layer, using the symbol VM to denote the set of all virtual machines in the virtual machine layer. The virtual machine $vm \in VM$ constitutes the environment for microservice operation. The leaf node of the tree topology is the microservice ms , which is scheduled by the container orchestration system to the corresponding vm .

III. SYSTEM MODEL AND PROBLEM FORMULATION

A. Service Profile

For an application built on microservice architecture, we describe its summary information and formalize the summary information as a service profile.

As the typical microservice architecture shown in Fig. 1, we represent the service profile as a tuple $\langle MS, MS_DEP \rangle$.

An application contains multiple microservices to provide services to users. We use the symbol MS to denote the set of all microservices contained in a microservice application. The total number of microservices is M . Since microservices constitute a directed graph, we use the symbol MS_DEP to describe the topological dependencies in a microservice application, and the set of MS_DEP contains the information of all edges in the directed graph. For example, if there is a call between microservice ms_i and ms_j , the invocation can be described as $(ms_i, ms_j, trans_{ij}) \in MS_DEP$, where ms_i denotes the caller, and ms_j denotes the callee. The traffic in the route is represented using $trans_{ij}$.

The service profile includes not only the topology diagram of the microservice, but also the resource requirements. We use REQ_i to describe the requirements of ms_i on physical resources (e.g., CPUs requirements, memory requirements, storage requirements).

B. Cluster Profile

Taking Amazon Cloud's EKS as an example, the profile of the container orchestration cluster is represented as $\langle N, LINK \rangle$, where the symbol N denotes the set of all working nodes in

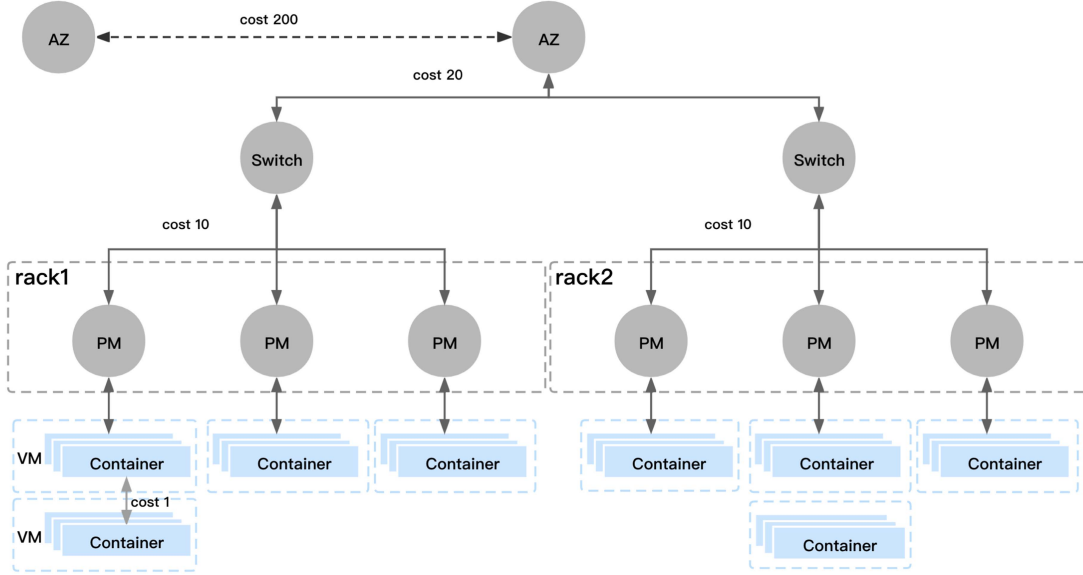


Fig. 3. Topology of cluster.

the container orchestration system. An element $n_x \in N$ denotes a working node. CAP_x and $ALLOC_x$ denote the total amount of resources and allocated resources in each dimension of node x . Respectively the symbol $LINK$ denotes the set of cluster communication links, where $link_l \in LINK$ constitutes the detailed description of the links. In the quadruplet $(ep_{l1}, ep_{l2}, cost_l, band_l)$, we use ep_{l1} and ep_{l2} to locate a link with two endpoints. Since different links have different communication costs and bandwidth, we use $cost_l$ to denote the communication overhead of the link and $band_l$ to denote the available bandwidth. The main notions are listed in Table I.

C. Optimization Objective

In this section, we take resource fragmentation, network overhead and network contention into account. And we build a multi-objective optimization model for microservices deployment.

Resource fragmentation is a key factor that affects the resource utilization of a cluster. The core purpose of our scheduling algorithm in this paper is to reduce the generation of resource fragments on work nodes. Therefore the mathematical model for the overall resource fragmentation of the cluster is defined as in (1)–(3). We use $exist_{i,x}$ to denote that service ms_i is deployed on node n_x . Since the service is sensitive to one resource but not to others, in R_x we assign different weights α to different resources, summing to 1.

$$FRAG = \sum_{n_x \in N} \sqrt{\frac{\sum_{d=1}^D (\gamma_x^d - R_x)^2}{D}} \quad (1)$$

$$\gamma_x^d = \frac{alloc_x^d + exist_{i,x} \cdot req_i}{cap_x^d} \quad (2)$$

$$R_x = \sum_{d=1}^D \frac{\alpha^d \gamma_x^d}{D} \quad (3)$$

In (4) we describe the cost of communication. (5) describes the interference of network, and in both equations $(ms_i, ms_j) \in DEP$, $link_l \in LINK(ms_i, ms_j)$, ms_j is the downstream service of ms_i .

$$COST = \sum_{(ms_i, ms_j) \in DEP} \sum_{link_l \in LINK} cost_l \quad (4)$$

$$INTER = \sum_{(ms_i, ms_j) \in DEP} \sum_{link_l \in LINK} \frac{trans_{ij}}{band_l} \quad (5)$$

D. Constraints

For the above three models, we want to achieve a low fragmentation rate, low communication overhead, low network interference, and optimize for these three objectives simultaneously. (6)–(10) are the constraints of our optimization.

$$alloc_x^d + req_i^d < cap_x^d, \forall n_x \quad (6)$$

$$\sum_{(ms_i, ms_j) \in DEP} trans_{ij} < band_l, \quad \forall link_l \in LINK(ms_i, ms_j) \quad (7)$$

$$exist_{i,x} = \begin{cases} balance, & \text{if } ms_i \text{ on } n_x, \\ 0, & \text{if } ms_i \text{ not on } n_x \end{cases} \quad \forall ms_i, \forall n_x \quad (8)$$

$$exist_{i,x} = \begin{cases} 1, & \text{if } ms_i \text{ on } n_x, \\ 0, & \text{if } ms_i \text{ not on } n_x \end{cases} \quad \forall ms_i, \forall n_x \quad (9)$$

$$\sum_{x=1}^N exist_{i,x} = 1, \forall ms_i \quad (10)$$

$$balance = \begin{cases} 1, & MAX\gamma_x - MIN\gamma_x \leq T, \\ 0, & MAX\gamma_x - MIN\gamma_x > T \end{cases} \quad \forall n_x$$

Eq. (6) restricts the amount of allocated resources from exceeding the maximum limit. (7) limits traffic from exceeding the maximum bandwidth. And (8) and (9) mean there will be

TABLE I
SYMBOLS IN THE DEPLOYMENT MODEL

Microservice	
MS	The set of all microservices contained in the application
MS_DEP	All dependencies between microservices
(ms_i, REQ_{ms_i})	$(ms_i, REQ_{ms_i}) \in MS$ indicates services ms_i and its resource requirements
$req_t^d \in REQ_i$	The demand for resource d by ms_i
$(ms_i, ms_j, trans_{ij})$	$(ms_i, ms_j, trans_{ij}) \in MS_DEP$ means ms_i invokes ms_j , and $trans_{ij}$ indicates the traffic generated by service call
$DM(ms_i) = ms_j$	ms_j is the upstream service of ms_i
Work Node	
N	The set of all work nodes
$(CAP_x, ALLOC_x)$	$n_x = (CAP_x, ALLOC_x) \in N$ describes the available and allocated resources of the work node n_x
cap_x^d	Total available resource of d on n_x
$alloc_x^d$	Allocated resource of d on n_x
Links	
$LINK$	The set of all link in the cluster
$(ep_{l_1}, ep_{l_2}, cost_l, band_l)$	$link_l = (ep_{l_1}, ep_{l_2}, cost_l, band_l) \in LINK$ describes the endpoints, cost and bandwidth of $link_l$

only one instance for one service, and it's located on node n_x . The (10) sets a threshold value T . Scheduling that breaks the threshold cannot take effect, which is to ensure the balance of node resource allocation and prevent the generation of large resource fragmentation.

$$\begin{aligned} & \text{Minimize } FRAG + COST + INTER \\ & \text{s.t. } Eq \cdot (6)-(10) \end{aligned} \quad (11)$$

The objective in (11) is to minimize the deployment fragments, cost, and interference under the constraints of (6)–(10).

IV. TOPOLOGY-AWARE SCHEDULING ALGORITHM

It is difficult to find the optimal solution to a multi-objective optimization problem directly, especially in the environment of cluster scheduling, which is affected by the dynamics of the environment and the results of scheduling decisions are effective. This means that the global optimal solution is often degraded by the changes of the environment. Therefore, this paper proposes a topology-aware scheduling strategy for microservices based on a heuristic graph partitioning algorithm. For the deployment of microservices, the policy only goes to a Best-Effort approach to find the optimal solution, in which the resource usage of each working node and the resource demand of microservices are considered to filter out the working nodes that are not suitable for the scheduling of that microservice. The the final scheduling

Algorithm 1: MOTAS Algorithm.

Require: Q : priority queue for applications;
 MS : microservice application to be scheduled;
 N : work nodes;
 DEP : dependencies of microservices;
 $LINK$: topology of nodes with cost and capacity;

- 1: **while** $Q \neq \emptyset$ **do**
- 2: $MS \leftarrow Q.pop()$
- 3: $p \leftarrow \text{recursiveGraphMapping}(MS, DEP, N, LINK)$
- 4: **if** $p \neq \emptyset$ **then**
- 5: $doPlacement(p)$
- 6: **else**
- 7: $Q.add(MS)$
- 8: **end if**
- 9: **end while**

location is confirmed by the evaluation of resource utilization, communication overhead, and network interference.

As shown in Algorithm 1, we maintain a priority queue for microservice applications, which supports the scheduling of microservice applications in the order of priority. We invoke a recursive mapping algorithm during the scheduling process to determine the mapping relationship from microservice applications to hosts. The recursive mapping algorithm will be described in Algorithm 2. In addition, we also introduce the idea of Gang Scheduling [23], where the actual execution of the container placement instruction is done only after all microservices in the application have been scheduled. This is because microservice applications usually contain many microservices; if scheduling is performed at the granularity of a single microservice, when some important microservices do not meet the scheduling conditions and cannot be deployed, other microservices will keep waiting for these services. The resources will be occupied for a long time, which results in a great waste of resources. Gang scheduling deploys microservices at the application granularity, which guarantees that all microservices of the application can be scheduled and deployed in a complete manner.

The design of Algorithm 2 is based on the idea of the heuristic Fiduccia Mattheyses algorithm [24], which continuously partition the set of working nodes by recursion to divide different network partitions and finally determines the placement of microservices. In the partitioning of the nodes, we consider the number of physical links and the communication cost between the worker nodes, aiming to reduce the communication overhead between different network partitions. In the specific design of the algorithm, two functions are called in each recursion. Among which $nodePartition()$ adopts the Fiduccia Mattheyses algorithm for partitioning the physical topology, and we add the consideration of link communication cost on the basis of dealing with the minimum partition, which requires the partition line to pass through edges with the minimum link communication cost.

However, since the topology of the working nodes is a typical tree structure, the partitioning process of the working nodes using this algorithm often leads to the microservices' aggregation. Therefore, in the division of microservice topology,

Algorithm 2: Recursive Mapping Algorithm.

Require: MS: microservice application to be scheduled;
 N: work nodes;
 DEP: dependencies of microservices;
 LINK: topology of nodes with cost and capacity;
Ensure: p: placements;

```

1: if |MS| = 0 then
2:   return nil
3: end if
4: if |N| = 1 then
5:   return p  $\leftarrow$  (MS, N)
6: end if
7: ( $N_0, LINK_0, N_1, LINK_1$ )
    $\leftarrow$  nodePartition(N, LINK)
8: result
    $\leftarrow$  svcPartition (MS,  $N_0, LINK_0, N_1, LINK_1$ )
9: if result = failed then
10:  return nil
11: end if
12: ( $MS_0, MS_1$ )  $\leftarrow$  result
13:  $p_0 \leftarrow$  MOTAS( $MS_0$ , DEP,  $N_0, LINK_0$ )
14:  $p_1 \leftarrow$  MOTAS( $MS_1$ , DEP,  $N_1, LINK_1$ )
15: return ( $p_0 + p_1$ )

```

resource utilization is also taken into account to ensure that the principle of balanced resource allocation is not violated in the case of single-point deployment.

The *svcPartition()* in Algorithm 2 is used to partition the microservice topology, but the partitioning and selection of microservices are more complex. This function will be described in detail in Algorithm 3. There are two stopping conditions for recursion: first, the MS set of the partitioned microservice group is the empty set, implying that the group of microservices is not involved in the deployment decision; second, the set N of the partitioned working nodes contains only one node, implying that the node is the node for service deployment.

Algorithm 3 first traverses the topology graph of microservice applications by means of hierarchical traversal, and schedules the microservices in the order of microservice application invocation to ensure that all other services that depend on the service have finished scheduling at the time of scheduling. The algorithm divides the microservice MS into two sub-partitions: MS_0 and MS_1 . Each of which can contain either partial services or all services. The tasks in MS_0 will be placed in N_0 , while the tasks in MS_1 will be placed in N_1 . In order to reduce the subsequent computation overhead, the working nodes that violate the resource balance provisions are first eliminated in the scheduling based on the resource demand information of the services. For the suitable nodes the minimum value of the obtainable communication cost and the corresponding working nodes are determined by the deployed services on them.

According to the optimization objectives in the deploying model, the measurement of resource fragmentation and network contention is also included in Algorithm 3. *getInter()* and *getFrag()* refer to the definitions given in (3) and (5),

Algorithm 3: Microservice Partition Algorithm.

Require: MS: microservice application to be scheduled;
 DEP: dependencies of microservices;
 N_0 : work nodes in part 0;
 N_1 : work nodes in part 1;
 $LINK_0$: links in part 0;
 $LINK_1$: links in part 1;
Ensure: (MS_0, MS_1): partition of microservices;

```

1: while MS  $\neq \emptyset$  do
2:  ms  $\leftarrow$  travallnOrder(MS)
3:  if isScheduled(ms) then
4:    continue
5:  end if
6:  ( $N'_0, LINK'_0, N'_1, LINK'_1$ )
    $\leftarrow$  filterBalanceNode( $N_0, LINK_0, N_1, LINK_1$ )
7:  ( $n_0, cost_0, n_1, cost_1$ )
    $\leftarrow$  getMinCost(ms, DEP,  $N'_0, N'_1, LINK'$ )
8:  ( $inter_0, inter_1$ )
    $\leftarrow$  getInter(ms, DEP,  $n_0, n_1, LINK'$ )
9:  ( $frag_0, frag_1$ )  $\leftarrow$  getFrag(ms,  $n_0, n_1$ )
10: if  $score_0 < score_1$  then
11:   $MS_0.add(ms)$ 
12: else
13:   $MS_1.add(ms)$ 
14: end if
15: end while
16: return ( $MS_0, MS_1$ )

```

and the suitability of each sub-partition can be evaluated by calculating the communication cost *cost*, network interference *inter* and fragmentation rate *frag*. Finally, the utility function *score()* is used to calculate the utility of deploying service *ms* in this partition, and *score()* is shown in (12), we still use α to denote the weight of each variable, and the sum of α is 1.

$$score = \alpha_c cost + \alpha_f frag + \alpha_i inter \quad (12)$$

V. TOPOLOGY-AWARE SCHEDULING FRAMEWORK

Due to the complexity of microservices, the topology is difficult to define and illustrate artificially in large microservice applications. In addition, receiving the influence of dynamism, the topology of microservice applications and the cluster environment may change and lead to the degradation of the quality of service of the application. We introduce a distributed tracking system to analyze the request information in the microservice system, obtain the execution graph of each request, and classify and synthesize the request execution graph according to the division of microservices to obtain the global topology description information of the microservice application. And we design the scheduling framework to respond to the quality of service degradation problem of the application.

A. Topology-Aware Scheduling Framework Architecture

It is very difficult to locate all kinds of performance bottlenecks in a microservice environments. To solve the dynamicity problem in microservice systems, distributed tracing systems are applied to analyze and monitor complex applications under distributed systems, as well as to locate the root causes that affect the performance of application services. The distributed tracing system can take the request as the tracing object and follow the execution path of the request to obtain the execution information of all services along the request execution path. This section introduces the distributed tracing system in the dynamic topology-aware scheduling framework for microservices with the following two main objectives:

First, acquire service profile dynamically: Microservice applications are characterized by a large number of services and complex invocation relationships and it is difficult to sort out the invocation relationships of large-scale microservices in practical scenarios. The way to define service profiles manually and input them into a topology-aware scheduling algorithm has greater limitations, both from the perspective of complexity and dynamism. By introducing a distributed tracking system into the dynamic topology-aware scheduling framework for microservices, the execution history of microservices can be analyzed by recording the execution paths of a large number of requests on the one hand, and the topology of microservices can be restored from the execution history graph.

Second, quantify the impact of cluster dynamics on microservice applications: the SLOs of microservice applications are determined by the execution time and communication overhead of individual microservices. While the cluster, as the bearer of a large number of microservices, has a highly dynamic resource environment and network environment. The SLOs of microservice applications have uncertainty in this situation. We implant probes of distributed tracing in microservices, analyze and extract the invocation path of each microservice under different requests. By analyzing the execution time of microservices at the nodes on the invocation path, we can effectively determine whether the microservices are affected by resource contention and bandwidth contention, and make timely adjustments based on the information of trace analysis.

As shown in Fig. 4, the design architecture of the dynamic topology-aware scheduling framework is presented. In this paper, we introduce a tracing module the scheduling framework. The design of the tracing module is mainly based on the implementation of Google's distributed tracing system Dapper [25] and its open-source version Jaeger and Zipkin: an observation probe is introduced in each microservice, the probe aggregates and reports the invocation information of each microservice instance to the tracing module. As shown in Fig. 5, the invocation information includes: request ID, trace ID, invoker ID, service name, invocation start time, invocation consume time, etc. The request ID is a unique identifier for the current request, and the request contains multiple calls; each call also has a unique identifier ID and contains information about the upstream call to analyze the dependency.

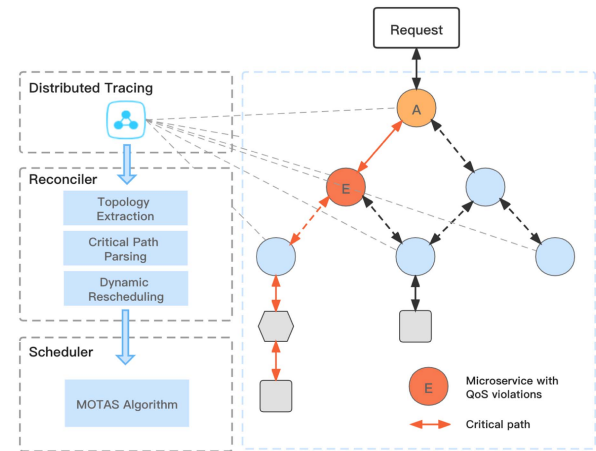


Fig. 4. Architecture of topology-aware scheduling framework.

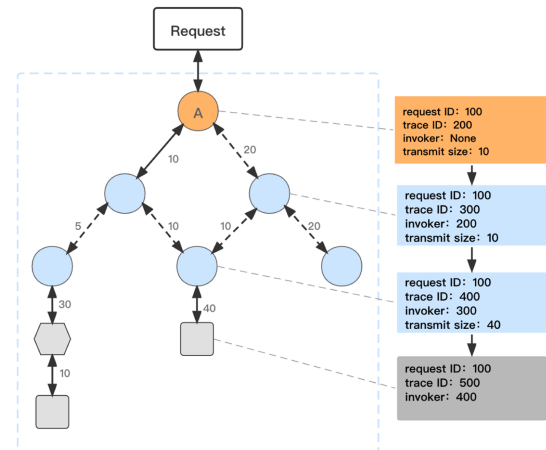


Fig. 5. Architecture of topology-aware scheduling framework.

The tracing module stores the invocation information in a separate database through the message middleware, which can relieve the writing pressure on the database caused by a large amount of call information to a certain extent. The topology of the entire microservice application can be obtained by synthesizing the invocation information. The graph database is also available as an option for the framework by storing the execution history graph in the graph database, allowing easy storage of complex invoker-server relationships between microservices based on request types, as well as efficient querying of the critical path/component extraction graph. In addition, in order to obtain information about the load on the invocation path, we additionally include information about the size of request packets and callback packets in the invocation information, which can be efficiently used to set the weights of each edge in the topology.

The reconciler in Fig. 4 is the core of the scheduling framework. The design concept of the reconciler is to keep the microservice application as a whole in an optimal state of service by adjusting microservices that are affected by resource contention

and degraded in quality of service. The parsing module in the reconciler continuously analyzes the request execution information reported by the distributed tracing system and restores the topology of the microservice application in a large number of request calls in terms of application as a category, which is stored in the form of a configuration file and becomes part of the summary description of the microservice application. In order to record the communication weights on each request link, the reconciler also categorizes the number and size of different requests during the period, and this information will also be stored in the form of a service profile, which is an important basis for the scheduler to make scheduling decisions.

In addition, the reconciler is also used to sense the performance of microservices. When the performance of microservices is affected by the resource occupation of working nodes or the bandwidth occupation of invocation links, the reconciler initiates a notification to the scheduler to inform it of the affected microservices. The scheduler adjusts the deployment of microservice applications by rescheduling to ensure the SLOs of time-sensitive applications, and the relevant algorithm and processing logic of the reconciler is further described in Sections V-B and V-C.

The scheduler in Fig. 4, as its name implies, is responsible for assigning work nodes to microservices. The scheduler supports the scheduling of general container applications, but can also receive the outline description of microservice applications and complete the scheduling of microservice applications based on the scheduling policy in Section V-C of this paper, in conjunction with the specific topology of the microservices.

It should be additionally noted that although the resource overhead and network footprint of the observation probe is relatively small, it may still have an impact on the performance of the microservice itself. In order to circumvent the impact of the probe on the application, we use the probabilistic sampler of Jaeger [26] and set the sampling frequency of this framework to 0.1 based on the known experimental user request rate, i.e., one-tenth of the calls are randomly collected and reported. It is experimentally observed that the loss to the microservice application throughput is less than 0.8% and the impact on latency is less than 0.5% when the sampling frequency is 0.1. However, there may be no way to use a probabilistic sampler as described above for a recently deployed application where the developers do not know the user request rate in a real-world environment. This needs to be adjusted according to the actual situation. For example, the larger the user request rate, the smaller the sampling frequency should be. To reduce the impact of trace sampling on microservice application throughput and response latency in a real-world environment, developers can also use a rate limiting sampler [27] or adaptive sampling [28], both of which are provided by Jaeger.

B. Extract Critical Path From the Trace

Applications designed with microservice architecture are dynamic, as shown in Fig. 6: (a) The updates to microservice application interfaces, changes to invocation methods and invocation relationships can lead to changes in a service profile,

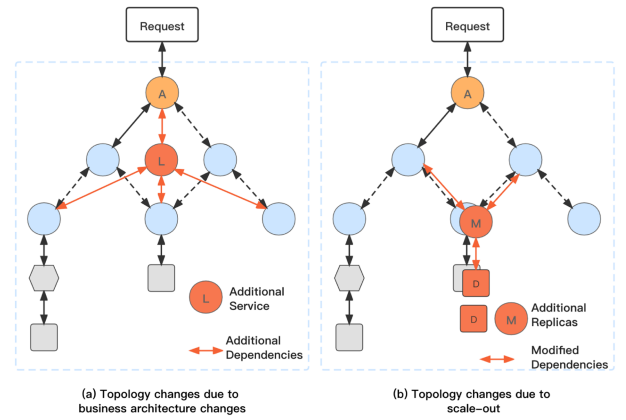


Fig. 6. The case of topology change.

and scheduling decisions that rely on application description information may be invalidated as a result; (b) The scaling strategies of container orchestration systems may increase or decrease the replicas of microservices, which may also influence the performance of static topology-aware scheduling algorithms.

Based on the analysis of microservice application invocation information with the tracing module, the topology diagram of the entire microservice application can be obtained. In some cases, service profile changes may not affect the overall performance of the application; it is only when the topology on the critical path changes that the overall performance of the application will be affected.

Fig. 7 shows that during the request-response process of a microservice application, there are one or more invocation paths to respond, where the path that has a decisive impact on the end-to-end response time of the request is defined as the “critical path (CP)”. Since the critical path is usually the longest path from the start of the client request to the return of the request by the microservice application, the critical path determines the end-to-end response latency of the microservice application. The request execution diagram shown in Fig. 7 shows all the execution paths of a request, and the invocation paths pass through the compose-service, user-service, media-service, text-service, user-mention-service, post-storage-service, url-shorten-service, user-timeline-service, home-timeline-service, etc. The invocation modes between services usually include the following three: serial invocation, parallel invocation, and asynchronous invocation.

The critical path determines the end-to-end response latency of the microservice application. Therefore, in order to measure the degree to which the microservice application is affected by the dynamics, it is necessary to analyze the critical paths of the different invocation interfaces of the microservice application and establish the mapping between the interfaces (usually in the form of “/api”) and the critical paths.

Algorithm 4 describes how to establish the corresponding critical paths for different requests. The critical path resolution algorithm starts from the root node of the request execution graph (i.e., the most upstream service nginx), and by checking the request execution graph, it obtains the last returned

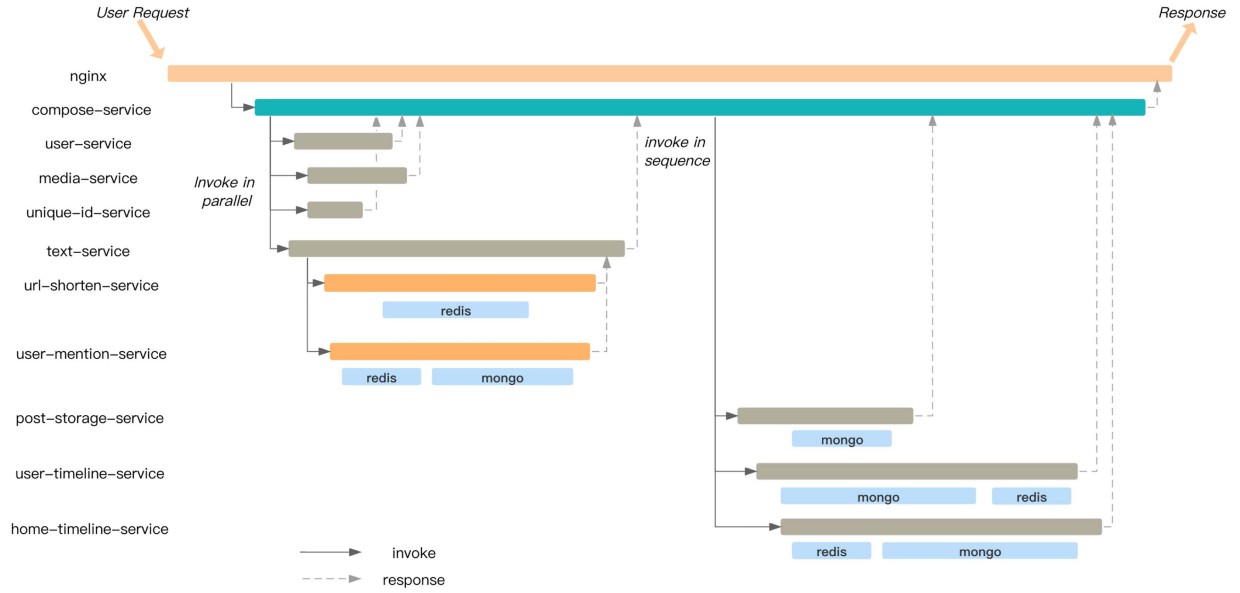


Fig. 7. Invocation of compose-post api in social network [16].

Algorithm 4: Critical Path Extraction Algorithm.

Require: Graph: execution graph of request;

UM: upstream service

Ensure: CP: critical path of request;

```

1: CP.add(UM)
2: instance ← UM
3: if instance.children = None then
4:   return CP
5: end if
6: DM ← UM.lastReturnedDM
7: while DM ≠ nil do
8:   CP.extend(CriticalPathExtraction(Graph, DM))
9:   DM ← DM.previousSerialInvoke
10: end while
11: return CP

```

microservice (home-timeline-service) among the downstream services. And then the algorithm recursively invokes the critical path resolution algorithm to analyze the critical path composition of the downstream service, and when the node to be searched is empty, the recursive algorithm returns part of the critical path. When the analysis of the home-timeline-service is completed, the same analysis will be performed on the previous microservice (i.e., text-service) that has a serial execution relationship with the home-timeline-service to obtain the partial critical path of this service. Finally, all the partial critical paths are combined to form the complete critical path.

From the critical path extraction algorithm, it can be found that the main basis for resolving the critical path is the invocation time and return time of each layer of microservices. So, it can be inferred that there are two main scenarios affecting the critical path of microservice applications:

First, the response latency of the critical path changes, but the composition of the critical path remains unchanged; this is due to the increase in the response latency of some services on the path, resulting in the overall response latency of the critical path.

Second, the response latency of the critical path changes because a microservice on the critical path is replaced by other services invoked in parallel, resulting in a change in the overall response latency.

The causes of the two cases are different, but the treatment is the same: locating the root node that produces the impact and executing a rescheduling policy for that node and its downstream services.

C. Dynamic Performance-Aware Scheduling

In this section, we propose algorithms to analyze the response latency of each service on the critical path and the total response latency of the critical path to locate the services that affect the application performance.

As shown in Algorithm 5, we divide the tracing data into multiple time slices for recording and analysis. In order to reduce the overhead caused by monitoring, we first analyze the delay of the critical path in each time slice. First, for the phenomenon of packet loss: request failure and timeout caused by the unstable network environment, the algorithm analyzes the response delay of T_{99} (latency for the 99th percentile) of the critical path and T_{50} (latency for the 50th percentile). When there is a tail latency in microservices, it means that the service quality of some users is not guaranteed, and the value of T_{99} is high, while the change of T_{50} is not significant, so the ratio of T_{99} to T_{50} can reflect the situation of tail latency better. When the service quality of most users is not guaranteed, both T_{99} and T_{50} are high, and their ratios cannot reflect the service quality, so we only use T_{50} to describe the quality of service. The performance evaluation function condition is T_{99}/T_{50} , which is used to quantify the

Algorithm 5: Performance-Aware Scheduling Algorithm.

Require: CP: critical path of request;
 Q: scheduling queue;
 N: work nodes;
 baseline: regular e2e latency of the request;
 threshold: regular ratio of T_{99}/T_{50}

Ensure: Θ ;

```

1: while true do
2:   slice  $\leftarrow$  tracer.getDate(CP.head)
3:    $T_{50} \leftarrow$  slice.percentile (50)
4:    $T_{99} \leftarrow$  slice.percentile (99)
5:   condition  $\leftarrow T_{99}/T_{50}$ 
6:   condition.getMovingAverage
7:   if condition > threshold or  $T_{50} > baseline * 2$  then
8:     for ms  $\in$  CP do
9:       ms $T_{50} \leftarrow$  msSlice.percentile (50)
10:      ms $T_{99} \leftarrow$  msSlice.percentile (99)
11:      condition  $\leftarrow$  ms $T_{99}/msT_{50}$ 
12:      if condition > threshold.ms or  $T_{50} > baseline.ms*2$  then
13:        Q.add(ms.subGraph)
14:        TopologyAwareScheduling(N)
15:        descheduling(ms.subGraph)
16:      end if
17:    end for
18:  end if
19: end while

```

quality of service, and when condition is greater than the set threshold, the microservice application is judged to be abnormal and the performance is degraded. We inject faults into the microservice application during the preprocessing phase and obtain the end-to-end response latency. Therefore, by analyzing these historical data, we can obtain the relationship between T50 and T99 in normal and abnormal states, and thus derive the T99/T50 thresholds. The threshold is set to 20 in this paper.

Another scenario is for the case of increased transmission delay due to restricted bandwidth or increased processing delay due to restricted computing resources. We select throughput as the metric and define the maximum throughput of application with condition. Algorithm 5 analyzes the average response delay T_{50} , and when its overall increase is more than twice, it means that most of the users' requests are received, which means that the service is considered to be affected.

For the affected service that experiences performance degradation, Algorithm 5 handles it by adding the service and its downstream services to the scheduling queue and assigning new hosts to them by rescheduling. So the performance of the whole application is brought to normal.

To reduce the monitoring overhead, Algorithm 5 only analyzes the latency of the critical path, and only when the critical path is abnormal, the nodes on the path are analyzed in turn. This is due to the fact that the invocation latency of the whole critical path determines the response latency of the whole application.

Although there is a scenario in which the critical path is replaced in some cases: the total response time on a certain invocation path exceeds the total response time of the critical path. For this scenario, Algorithm 5 can still handle it because the total response latency of the application also changes at this time, which can trigger the response mechanism of the algorithm to add the exception service and downstream services to the scheduling queue.

In addition, since the response latency is not stable, the monitoring process may have a big difference in the data of each different time slice. In order to prevent the occasional rise in latency from triggering the response of rescheduling resulting in frequent rescheduling of the microservice container, we also add the moving average algorithm (*getMovingAverage()*) to calculate the request latency within a certain time period and predict the long-term trend of latency change, which is shown as (13).

$$delay_t = \beta delay_{t-1} + (1 - \beta) delay \quad (13)$$

The sampling delay obtained for the current time slice period is recorded as $delay$, while $delay_t$ represents the moving average experiment for the current time slice period. The value of $delay_t$ is determined by both the sampling coefficient and the current sampling delay. We set β as the coefficient to control the number of time slices involved in the moving average calculation. The β is set to 0.9 in this paper, which means we take the last 10 values into account. The moving average method is also used to sample and evaluate *condition*, *condition* is one of the metrics to judge the quality of service.

In the final part of the algorithm, when it is determined that the performance of a microservice is dynamically affected, the impact on that microservice will be eliminated by adjusting the node deployment location based on the dynamic response policy in Section IV.

VI. PERFORMANCE EVALUATION

In this section, we will use an open-source benchmark suite DeathStarBench [16] to evaluate the framework. We will compare the framework with Kubernetes [29] (predecessor as [30]) default scheduling policy in terms of resource utilization and quality of service, and validate the response of the framework to dynamic influences.

A. Evaluation Setup

The experimental platform in this paper is built with OpenStack (Train Version) [31], an open-source cloud computing management platform. 10 physical machines are included in the cloud platform, each using Intel x86 Xeon E5 series processors, containing 16 to 32 CPU cores, and equipped with 128 GB of physical memory. The framework of the prototype system uses Kubernetes (v1.19.2) as the underlying support, and the algorithm in this paper is accomplished with its scheduler framework. The entire container platform contains one control node and eight worker nodes, with the control node configured with 16 CPU cores and 32 GB of memory. The worker nodes are configured with 8 CPU cores and 16 GB of memory distributed on

different physical machines. We also use Linux Traffic Control as an injection tool for network exceptions.

The evaluation is conducted through two end-to-end interactive and responsive real-world microservice applications: Social Network and Hotel Reservation, both provided by the DeathStar-Bench suite [16]. Social Network implements a broadcast-style social network whereby users can publish, read, and react to social media posts. It contains 36 unique microservices. And Hotel Reservation is an online hotel reservation site for browsing hotel information and making reservations with 15 unique microservices. In addition, these services are deployed in separate Docker containers.

To test the performance of the scheduling framework, we also use the following tools:

- *wrk2* [32]: To accurately evaluate the performance of microservice applications under different deployment strategies, this paper uses the open-source workload generator *wrk2* to simulate real requests. *wrk2* is a popular HTTP benchmarking tool that runs on a single multi-core CPU and generates an open-loop load in the form of a large number of HTTP requests to the web application (open-loop load means that all requests are entered at the scheduled time, without waiting for previous requests to be responded to. Without waiting for previous requests to be answered, the open-loop load reflects the true performance of the server and the actual latency of operation). The load generator itself can generate load in the form of rated value or obey Poisson distribution, and simulate the interaction from client to microservice application in the form of multiple threads and multiple connections. In order to simulate the load environment under real scenarios to the greatest extent, this paper uses real user traffic as the input load to simulate the pressure on microservice applications under different scenarios.
- *Linux tc* [33]: In the testing of microservice applications, the available bandwidth and response latency of containers also affect the end-to-end response time and throughput of microservice applications. When the import or export bandwidth of a worker node is exhausted, the response of containers on that node will be greatly affected. In order to simulate real application scenarios especially the large-scale offline operation scenario in the mixed part scenario, and evaluation of the impact of high bandwidth consumption on the performance of microservice applications, this paper adopts the *Trickle* tool under Linux to control the available bandwidth and latency of the working nodes to simulate the abnormal conditions on the link.

B. Evaluation Results

1) *Resource Utilization*: In this evaluation, we generate container loads with different resource requirements based on Alibaba's public trace [6], and compares the maximum number of containers that can be hosted under the same resource environment. As shown in Fig. 8 ("ratio" indicates the ratio of the maximum number of containers obtained by the method used to the number obtained by the First Fit), MOTAS provides a small improvement in resource utilization compared to the

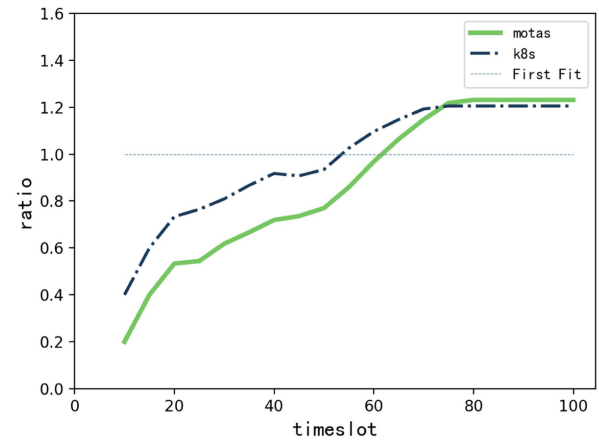


Fig. 8. Utilization of MOTAS and K8s.

default scheduling policy of Kubernetes. Also, it can be observed that the scheduling overhead of MOTAS is a bit higher, but it is still acceptable.

2) *Quality of Service*: As shown in Fig. 9, we plot a comparison of the cumulative distribution function (CDF) of the end-to-end latency for different applications. Since social network applications are more complex, we test them with different interfaces. The comparison shows that the MOTAS can provide lower end-to-end response latency for microservice applications compared to the Kubernetes default scheduling policy. Observing the performance of home-timeline interface, user-timeline interface, and hybrid interface in hotel reservation application, the topology-aware scheduling strategy has lower average response latency and performs better on the metric of T_{99} , which means that MOTAS also has an advantage in tail latency control.

We test the throughput of the Hotel Reservation. The workload is generated by *wrk2* with different interfaces. We continuously increase the workload, record the variation of the T_{50} and T_{99} latency of the application, and calculate the ratio of T_{99}/T_{50} (the maximum throughput is considered to be reached when T_{99}/T_{50} is more than 10). The application keeps the allocated resources unchanged and does not enable automatic scaling. The results obtained from several experimental tests are shown in Fig. 10 (we use the vertical axis label "Latency" to indirectly state "Throughput," i.e., the higher the latency, the lower the throughput). The Kubernetes default scheduling policy reaches the maximum throughput at 10 times the workload, and the MOTAS can provide a throughput improvement of about 20%.

3) *Dynamic Violation Reconciliation*: To verify the ability of MOTAS to handle QoS violations, we set send delay (10 ms) and generate randomly(1%) packet loss (1%) to simulate QoS violations on the VM where the text-service is located through *tc*. As shown in Figs. 11 and 12, we compare the latency of all services of the compose-post interface of the Social Network. It can be found that the injected network delay and packet loss cause an increase in the overall end-to-end latency. Our framework is able to identify and handle the violation, and service quality is restored by rescheduling text-service and its downstream services.

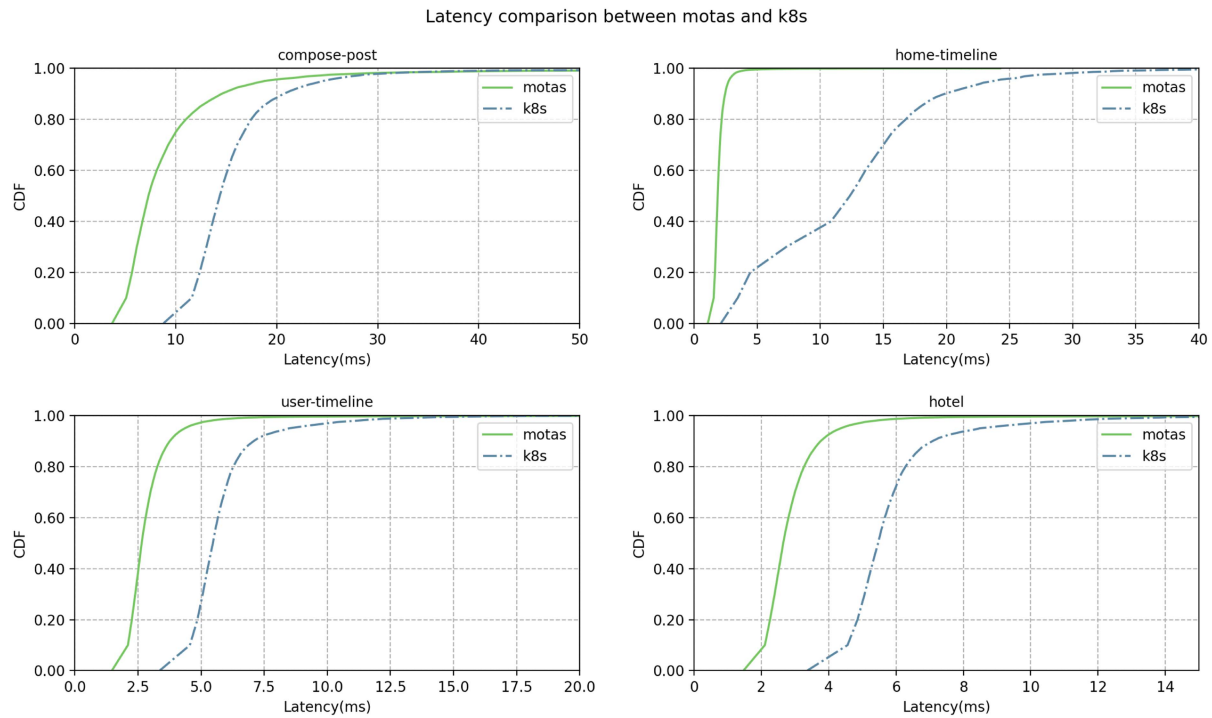


Fig. 9. E2E latency of MOTAS and K8s.

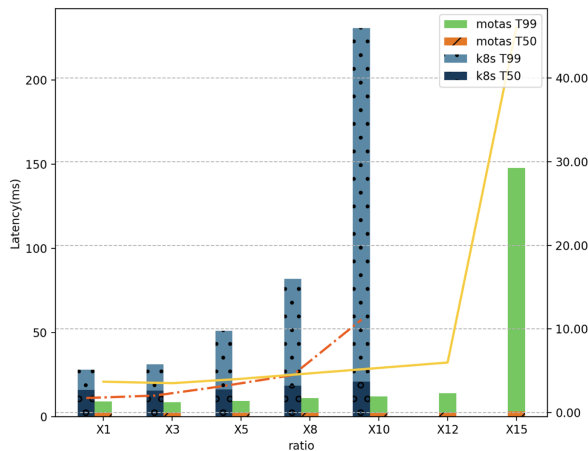


Fig. 10. Throughput of MOTAS and K8s.

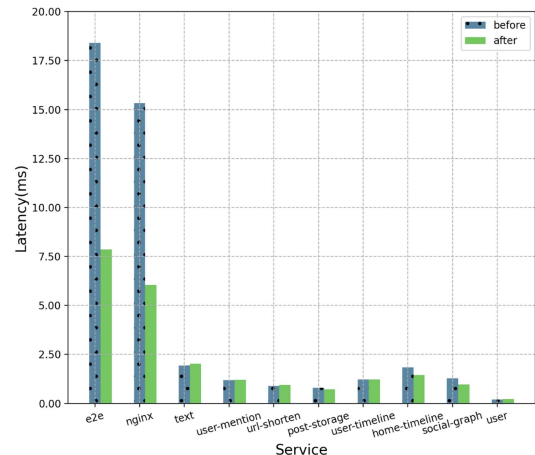


Fig. 11. Injecting delay.

VII. RELATED WORK

A. Microservice Characteristics

Several research works compared the performance of monolithic and microservice architectures and discovered that while microservices have some advantages over monoliths in terms of lower infrastructure costs, but microservices have significant overhead in terms of programming language runtime, cache hit rate reduction, network virtualization, microservice communication, etc [16], [20], [34], [35].

The study conducted in [6] using Alibaba public datasets showed that the invocation dependency graph of microservices is highly dynamic in production environment. And

Gan et al. [16] investigated microservice architecture characteristics and found that the performance of microservice applications is often defined by their topology, thus the latency requirements for each layer of services are much more stringent than for typical monolithic applications.

B. Container Orchestration

McDaniel et al. [36] investigated the interference between multiple applications placed in containers sharing underlying resources and found that I/O bandwidth contention brings some impact on container application performance. Running microservice applications in containers puts more pressure on

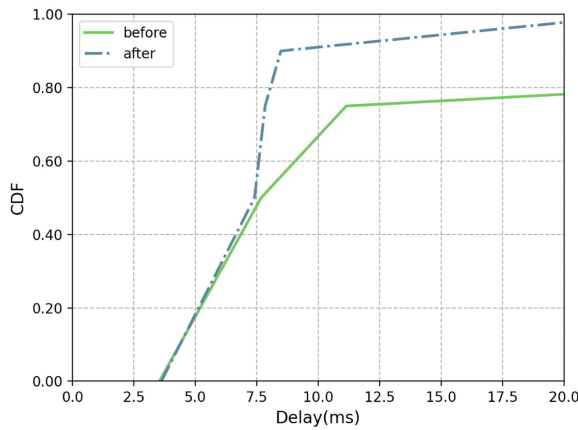


Fig. 12. Injecting packet loss.

system resources than traditional monolithic applications, and the infrastructure faces new challenges in terms of resource management, scheduling, monitoring and observability. The orchestration system based on Borg's open source implementation evolved into Kubernetes, which has since become the de facto standard for distributed container orchestration, as container orchestration system continue to advance. However, the container operating environment is highly dynamic and complex, with SLOs that need to be met in SLAs on the one hand, and cloud service providers needing to minimize costs on the other, but current orchestration systems only provide simple scheduling policies, and there is an urgent need for more complex policies to satisfy both the demands for quality of service and cost reduction requirements.

Guan et al. [37] designed an application-oriented container resource allocation architecture with the goal of minimizing the cost of application deployment in a distributed system while supporting automatic scaling as the workload of cloud applications changes. Zhang et al. [38] proposed a two-phase scheduling strategy that considers both container and virtual machine placement, but with a single optimization goal aimed at improving the utilization of physical resources. Adam et al. [39] proposed two-stage stochastic programming resource allocator (2SPRA), which explored resource allocation optimization for the multi-tier architecture of microservice applications with the aim of reducing the response latency of microservice applications. However, the objectives of these methods are relatively single.

In addition, for the need of monitoring and observation in complex distributed systems, Sigelman et al. [25] proposed a low-overhead, non-intrusive, distributed deployment tracing framework that can effectively provide application-oriented tracing capability, which can provide good support when studying the microservice container orchestration problem in complex distributed environments.

C. Resource Management and Task Scheduling

Different optimization goals for this problem lead to different optimization ideas, including response time [40], budget cost [41], security [42], etc. Among the various optimization

objectives, performance and resources are the most concerned, e.g., fair scheduling strategies [7], [43], [44] were used to guarantee the performance of applications, but they perform poorly in terms of resource utilization. Bin packing [8] was used to improve the resource utilization but it does not guarantee the quality of service. Heuristic resource allocation algorithms [45], [46] considered objectives in terms of user budget cost, makespan, total cost of the tasks, system load, etc. These approaches are able to obtain a feasible solution for multiple optimization objectives by pre-defining reasonable heuristics and continuously tuning them, but scheduling of online services running in the form of containers is not considered in these works.

Kaewkasi et al. [47] proposed an ant colony optimization algorithm for scheduling containers, the work focused on resource utilization of the cluster, but the limitation is that only resources are evaluated and the relationship between resources and quality of service is avoided. Mao et al. [48] was more concerned with balanced scheduling of resources within a cluster. Guerrero et al. [49] used genetic algorithms to optimize resource allocation and elasticity management of containers, but this work ignored the fact that in the microservice form, there is a lot of inter-communication between containers and the impact that these communication overheads bring to the microservice application as a whole.

Another trend is the application of machine learning methods to resource management and scheduling. Mao et al. [50] applied reinforcement learning to the problem of multi-resource packing for batch jobs and achieved good results in improving resource utilization. Hou et al. [51] combined feature inference approach to deal with the problem of microservice task scheduling, the algorithm improves the efficiency of microservices under constrained resources by learning microservice features and generating specific resource management policies, but the approach requires a large amount of trace data, and it cannot handle well microservice tasks with high variability and dynamics in multi-tenant and multi-task cluster scenarios. Auto-pilot [52] combined time series analysis and reinforcement learning algorithms to scale the number of containers and related resources, but the drawback is that independent scaling of each container may yield locally optimal results when the application has large-scale and complex dependencies. The machine learning approach usually achieves good results in resource management, but considering the dynamic changing characteristics of microservice systems in the actual production environment may lead to repeated model reconstruction efforts, such methods require continuous investment of large amounts of manpower to physically train high-performance models, which does not meet the scheduling requirements of container orchestration systems in production environments.

VIII. CONCLUSION

In this paper, we present a topology-aware placement framework for microservice applications scheduling in the cloud environment. The foundation of this framework is the graph mapping algorithm based on the topologies of the cluster and microservices. It is shown that the algorithm can reduce network communication overhead of microservice applications

and avoid network interference while ensuring cluster resource utilization. Based on the open-source microservices benchmark, we evaluated the framework against the Kubernetes default scheduling policy. And the evaluations effectively demonstrate that the framework can achieve high resource utilization while significantly reducing the end-to-end latency and increasing the throughput at QoS. However, there are shortcomings in our current work. The scale of the tests does not reach the true cloud scale, so the proposed framework cannot be directly applied to the real-world production environment without modification. In future work, we will further expand the cluster scale, microservice scale, and input workload scale.

REFERENCES

- [1] J. Thönes, "Microservices," *IEEE Softw.*, vol. 32, no. 1, pp. 116–116, May/Jun. 2018.
- [2] D. Merkel et al., "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, pp. 2, 2014.
- [3] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization Versus containerization to support PaaS," in *Proc. IEEE Int. Conf. Cloud Eng.*, 2014, pp. 610–614.
- [4] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. Philadelphia, PA, USA: O'Reilly Media, Inc., 2016.
- [5] A. De Iasio, A. Futno, L. Goglia, and E. Zimeo, "A microservices platform for monitoring and analysis of IoT traffic data in smart cities," in *Proc. IEEE Int. Conf. Big Data*, 2019, pp. 5223–5232.
- [6] S. Luo et al., "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 412–426.
- [7] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *Proc. 8th USENIX Symp. Netw. Syst. Des. Implementation*, 2011, pp. 1–14.
- [8] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 4, pp. 455–466, 2014.
- [9] M. Ganguli et al., "CPU overprovisioning and cloud compute workload scheduling mechanism," Mar. 20, 2018, US Patent 9,921, 866 B2.
- [10] S. A. Jyothi et al., "Morpheus: Towards automated SLOs for enterprise clusters," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 117–134.
- [11] A. U. Gias, G. Casale, and M. Woodside, "ATOM: Model-driven autoscaling for microservices," in *Proc. IEEE 39th Int. Conf. Distrib. Comput. Syst.*, 2019, pp. 1994–2004.
- [12] M. Lin, J. Xi, W. Bai, and J. Wu, "Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud," *IEEE Access*, vol. 7, pp. 83088–83100, 2019.
- [13] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 805–825.
- [14] T. Zheng et al., "SmartVM: A SLA-aware microservice deployment framework," *World Wide Web*, vol. 22, no. 1, pp. 275–293, 2019.
- [15] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.
- [16] Y. Gan et al., "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proc. 24th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2019, pp. 3–18.
- [17] C. Lu, K. Ye, G. Xu, C.-Z. Xu, and T. Bai, "Imbalance in the cloud: An analysis on alibaba cluster trace," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 2884–2892.
- [18] L. A. Barroso and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synth. Lectures Comput. Architecture*, vol. 4, no. 1, pp. 1–108, 2009.
- [19] H. K. Ala'a Al-Shaikh, A. Sharih, and A. Sleit, "Resource utilization in cloud computing as an optimization problem," *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, no. 6, pp. 336–342, 2016.
- [20] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *Proc. IEEE Int. Symp. Workload Characterization*, 2016, pp. 1–10.
- [21] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Comput. Archit. Lett.*, vol. 17, no. 2, pp. 155–158, Jul.-Dec. 2018.
- [22] EKS Amazonnetworking, Accessed: Dec. 20, 2021. [Online]. Available: <https://docs.aws.amazon.com/eks/latest/userguide/eks-networking.html>
- [23] D. G. Feitelson, "Packing schemes for gang scheduling," in *Proc. Workshop Job Scheduling Strategies Parallel Process.*, Springer, 1996, pp. 89–110.
- [24] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. IEEE 19th Des. Automat. Conf.*, 1982, pp. 175–181.
- [25] B. H. Sigelman et al., "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.
- [26] Probabilistic sampler, Accessed: Dec. 15, 2021. [Online]. Available: <https://www.jaegertracing.io/docs/1.40/sampling/#client-sampling-configuration>
- [27] Rate limiting sampler, Accessed: Dec. 15, 2021. [Online]. Available: <https://www.jaegertracing.io/docs/1.40/sampling/#client-sampling-configuration>
- [28] Adaptive sampling, Accessed: Dec. 15, 2022. [Online]. Available: <https://www.jaegertracing.io/docs/1.40/sampling/#adaptive-sampling>
- [29] Kubernetes, Accessed: Dec. 20, 2021. [Online]. Available: <https://kubernetes.io>
- [30] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, pp. 1–17.
- [31] Openstack, Accessed: Dec. 20, 2021. [Online]. Available: <https://www.openstack.org>
- [32] wrk2, Accessed: Dec. 20, 2021. [Online]. Available: <https://github.com/giltene/wrk2>
- [33] Linux Traffic Control, Accessed: Dec. 20, 2021. [Online]. Available: <https://man7.org/linux/man-pages/man8/tc.8.html>
- [34] M. Villamizar et al., "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Proc. IEEE 10th Comput. Colombian Conf.*, 2015, pp. 583–590.
- [35] M. Villamizar et al., "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures," in *Proc. IEEE/ACM 16th Int. Symp. Cluster Cloud Grid Comput.*, 2016, pp. 179–182.
- [36] S. McDaniel, S. Herbein, and M. Taufer, "A two-tiered approach to I/O quality of service in docker containers," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2015, pp. 490–491.
- [37] X. Guan, X. Wan, B.-Y. Choi, S. Song, and J. Zhu, "Application oriented dynamic resource allocation for data centers using docker containers," *IEEE Commun. Lett.*, vol. 21, no. 3, pp. 504–507, Mar. 2017.
- [38] R. Zhang et al., "Container-VM-PM architecture: A novel architecture for docker container placement," in *Proc. Int. Conf. Cloud Comput.*, Springer, 2018, pp. 128–140.
- [39] O. Adam, Y. C. Lee, and A. Y. Zomaya, "Stochastic resource provisioning for containerized multi-tier web services in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 7, pp. 2060–2073, Jul. 2016.
- [40] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 9, pp. 2114–2129, Sep. 2019.
- [41] W. Zheng and R. Sakellariou, "Budget-deadline constrained workflow planning for admission control," *J. Grid Comput.*, vol. 11, no. 4, pp. 633–651, 2013.
- [42] F. Abazari, M. Analoui, H. Takabi, and S. Fu, "MOWS: Multi-objective workflow scheduling in cloud computing based on heuristic algorithm," *Simul. Modell. Pract. Theory*, vol. 93, pp. 119–132, 2019.
- [43] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Princ.*, 2009, pp. 261–276.
- [44] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.
- [45] L. Zuo, L. Shu, S. Dong, C. Zhu, and T. Hara, "A multi-objective optimization scheduling method based on the ant colony algorithm in cloud computing," *IEEE Access*, vol. 3, pp. 2687–2699, 2015.
- [46] Q. Guo, "Task scheduling based on ant colony optimization in cloud environment," in *Proc. AIP Conf. Proc.*, vol. 1834, no. 1, AIP Publishing LLC, 2017, Art. no. 040039.

- [47] C. Kaewkasi and K. Chuenmuneewong, "Improvement of container scheduling for docker using ant colony optimization," in *Proc. IEEE 9th Int. Conf. Knowl. Smart Technol.*, 2017, pp. 254–259.
- [48] Y. Mao, J. Oak, A. Pompili, D. Beer, T. Han, and P. Hu, "DRAPS: Dynamic and resource-aware placement scheme for docker containers in a heterogeneous cluster," in *Proc. IEEE 36th Int. Perform. Comput. Commun. Conf.*, 2017, pp. 1–8.
- [49] C. Guerrero, I. Lera, and C. Juiz, "Genetic algorithm for multi-objective optimization of container allocation in cloud architecture," *J. Grid Comput.*, vol. 16, no. 1, pp. 113–135, 2018.
- [50] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.
- [51] X. Hou et al., "AlphaR: Learning-powered resource management for irregular, dynamic microservice graph," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2021, pp. 797–806.
- [52] K. Rzaqca et al., "Autopilot: Workload autoscaling at Google," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.



Xin Li (Member, IEEE) received the BS and PhD degrees from Nanjing University, in 2008 and 2014, respectively. Currently, He is an associate professor in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. His research interests include distributed computing, cloud computing and data management.



Junsong Zhou (Student Member, IEEE) received the BS degree from Huazhong University of Science and Technology, in 2019. He is currently working toward the MS degree with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA). His research interests include distributed system and cloud computing.



Xin Wei (Student Member, IEEE) is currently working toward the BS degree with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA). His research interests include distributed system, cloud computing, and service mesh.



Dawei Li (Member, IEEE) received the bachelor's degree from the Department of Electronics and Information Engineering (currently, School of Electronic Information and Communications), Huazhong University of Science and Technology, Wuhan, China, in 2011, and the PhD degree from the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA, in 2016. He is currently an assistant professor with the Department of Computer Science, Montclair State University, Montclair, NJ, USA. He has published research works in the *IEEE INFOCOM*, the *IEEE Transactions on Parallel And Distributed Systems*, and the *IEEE Transactions on Computers*. His research interests include the general fields of parallel and distributed systems, including green computing, data center networks, and cloud and edge computing.



Zhuzhong Qian (Member, IEEE) received the PhD degree in computer science, in 2007. He is currently a professor with the Department of Computer Science and Technology, and a member with the National Key Laboratory for Novel Software Technology, Nanjing University, China. He has authored or coauthored research papers in journals, such as *Transactions on Parallel and Distributed Systems*, *Transactions on Networking*, *Transactions on Computers*, and *Transactions on Mobile Computing*, and conferences, such as INFOCOM, ICDCS, SECON, and IPDPS. His research interests include cloud computing, edge computing, and distributed machine learning. He is the chief member of several national research projects on cloud computing and edge computing. He was the recipient of the best paper awards from IMIS 2013, ICA3PP 2014, and APNet 2018.



Jie Wu (Fellow, IEEE) is the director of the Center for Networked Computing and Laura H. Carnell professor with Temple University. He also serves as the director of International Affairs with the College of Science and Technology. He served as Chair of Department of Computer and Information Sciences from the summer of 2009 to the summer of 2016 and associate vice provost for International Affairs from the fall of 2015 to the summer of 2017. Prior to joining Temple University, he was a program director with the National Science Foundation and was a distinguished Professor with Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, network trust and security, distributed algorithms, applied machine learning, and cloud computing. He regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including *IEEE Transactions on Mobile Computing*, *IEEE Transactions on Service Computing*, *Journal of Parallel and Distributed Computing*, and *Journal of Computer Science and Technology*. He is/was general chair/co-chair for IEEE IPDPS'08, IEEE DCOS'09, IEEE ICDCS'13, ACM MobiHoc'14, ICPP'16, IEEE CNS'16, WiOpt'21, and ICDCN'22 as well as program chair/cochair for IEEE MASS'04, IEEE INFOCOM'11, CCF CNCC'13, and ICCCN'20. He was an IEEE Computer Society distinguished visitor, ACM distinguished speaker, and chair for the *IEEE Technical Committee on Distributed Processing (TCDP)*. He is a Fellow of the AAAS. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.



Xiaolin Qin (Member, IEEE) is a professor in the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics. His research interests include data management and knowledge discovery.



Sanglu Lu (Member, IEEE) received the BS, MS, and PhD degrees in computer science from Nanjing University, in 1992, 1995, and 1997, respectively. She is currently a professor with the Department of Computer Science and Technology and the State Key Laboratory for Novel Software Technology. She has authored or coauthored more than 80 papers in refereed journals and conferences. Her research interests include distributed computing, wireless networks, and pervasive computing.