# 基于多样性的微服务韧性部署机制 *



余 航, 许 博†, 王秀磊
(陆军工程大学 指挥控制工程学院, 南京 210007)

**摘 要**: 针对微服务架构软件系统的共享漏洞问题,面向微服务下的部署工作进行了研究,提出了一种面向韧性抗毁的多样性微服务动态部署策略,能够利用多样性部署特点,根据资源约束情况部署多样化的微服务组件,兼顾集中式部署与负载均衡的优势,有效缓解了同构性带来的问题,增强了软件系统的韧性能力。在此基础上实现了一种最小负载部署算法(Loadc-Min),并通过实验和六种算法进行了比较,实验结果显示 Load-Min 部署算法相较其他经典算法,在资源利用效能、系统安全性等方面均有较大的提升。

**关键词**: 微服务; 多样性; 软件系统韧性; 容器部署; 服务组合
**中图分类号**: TP393.0 **文献标志码**: A **文章编号**: 1001-3695(2022)03-035-0845-06
doi:10.19734/j.issn.1001-3695.2021.08.0347

# Diversity-based microservice resilience deployment mechanism

Yu Hang, Xu Bo†, Wang Xiulei
(*Command & Control Engineering College, Army Engineering University of PLA, Nanjing* 210007, *China*)

**Abstract**: In order to alleviate the homogeneity problem, this paper proposed a diversity-based microservice resilience deployment mechanism. It designed the strategy to deploy a wide variety of instances according to resource constraints. Combined with centralized deployment and load balancing, the proposed strategy could effectively alleviate the homogeneity problem and enhance the resilience of the system. Thus, it implemented a minimum load deployment algorithm (Load-Min). Simulation experiments show that compared with other six classical algorithms, Load-Min algorithm has great improvement in system security, resource utilization, and load balancing performance.

**Key words**: microservice; diversity; resilience of software systems; container deployment; service composition

## 0 引言

系统维持服务持续运行、确保在面临多种异常情况下完成关键任务的能力通常被称为系统韧性[1]。为增强软件系统韧性,服务提供商通常会采取多样性及冗余部署的方式[2]。传统软件系统的单体架构要求软件必须整体部署[3],这使得实现多样性和冗余技术将付出成倍的代价[4]。而事实上,软件系统不同模块的重要性及脆弱性也不尽相同,通过增强部分模块的多样性即可有效提升系统整体的韧性抗毁能力。

微服务架构[5]可以解决当前单体架构所面临的问题,由于微服务架构下软件系统被高度解耦[6],使得微服务组件能够利用容器镜像进行开发,提高了微服务组件的可重用性,所以微服务架构下的冗余部署相较于单体架构更加灵活高效。但同时高可重用性同时也意味着基于同一镜像部署的多个组件容器都是同构的。现有研究表明,容器镜像中普遍存在安全漏洞[7]。因此一旦攻击者获得目标系统中某一微服务组件的漏洞,那么攻击者随后将反复利用这一漏洞成功攻击使用同样镜像的容器,从而严重影响系统提供服务能力。当前研究人员普遍认为,采用异构部署[8]等多样性方法是解决同构性问题的有效手段[2]。对于采取了多样性的系统而言,攻击者只有成功获取了每种版本实现的漏洞才有可能在攻击中取得成功;并且即便攻击者获得了暂时的成功,这一成功也是难以持续的。因此,版本数越多,多样性指标越高,攻击者的攻击就越难以获得成功。但多样性指标并非越高越好,不同微服务之间、以及同一微服务的不同版本实现在资源消耗、负载情况、故障概率等方面不尽相同,所以多样性的应用将进一步扩大系统的复杂性,从而在部署、运维、管理等方面带来了新的挑战。

本文旨在解决微服务架构下软件系统多样性部署问题,实现在异构的基础设施上部署异构容器,有效增加软件系统的多样性,使得攻击者无法利用容器的同构性,对微服务架构的软件系统进行大规模破坏和打击,从而保证软件系统的韧性抗毁能力。

本文的主要贡献包括:a)建立了一种多样化微服务的部署模型,以最大化系统的韧性抗毁能力;b)提出了一种基于多样性的异构容器编排方案,较好的克服了共享漏洞带来的问题,通过冗余部署并提高多样性水平,使得微服务可用性大大增强;c)提出了一种最小负载部署算法(Load-Min),该算法结合了集中式部署和分散部署的优势,既通过集中式部署确保了较低的链路负载,又通过对多版本实现的分散部署避免了服务请求对单个节点的过度依赖,同时兼顾了负载均衡。

## 1 相关工作

目前,在微服务架构下的服务链部署问题上,已经有学者提出了一些切实可行的解决方案,但主要还是从效率出发开展研究。Wan 等人[9]利用 Docker 容器分层的特点提出了一种资源分配算法 EPTA(EC placement and task assignment algorithm),该算法通过平衡唤醒成本、安装成本和通信成本,显著降低了部署成本。谷允捷等人[10]提出了一种基于重叠网络结构的服务功能链时空优化编排策略,该策略综合考虑了节点计

**收稿日期**: 2021-08-01; **修回日期**: 2021-09-28 **基金项目**: 国家自然科学基金资助项目;国家博士后科学基金资助项目;江苏省自然科学基金青年基金资助项目

**作者简介**:余航(1997-),男,浙江衢州人,硕士,主要研究方向为网络安全、软件定义网络、云计算;许博(1980-),男(通信作者),甘肃兰州人,副教授,硕导,博士,主要研究方向为网络性能测量、网络流量识别、软件定义网络、信息系统开发等(xubo820@163.com);王秀磊(1988-),男,山东济宁人,讲师,博士,主要研究方向为边缘计算、网络功能虚拟化、网络安全、网络管理.

算资源、网络链路资源与时延约束,将编排问题转换为最短路径问题,并用模拟退火算法进行了求解。Han 等人[11]针对工作负载进行性能分析实验,从而得到精细的资源需求,并基于贪婪的启发式算法执行微服务部署,该算法创新之处在于通过使用从剖析结果得出的精细化资源需求来考虑应用程序性能,但该算法只考虑了单条服务链的情况。Joseph 等人[12]提出了一种新颖的启发式方法 IntMA(interaction-aware microservice allocation),以借助从交互图获得的交互信息,从而以交互感知的方式部署微服务。但以上关于部署问题的工作都未基于多样性进行过探讨。

在引入多样性的同时,如何在服务器集群中部署多版本的微服务组件,从而使得系统更安全高效,成为了研究人员需要关注的问题。目前部分工作结合了多样性进行考虑,Torkura 等人[13]首次将移动目标防御技术(moving target defense,MTD)应用到微服务架构上,利用 MTD 机制,通过多样化部署的安全策略最小化共享漏洞,为攻击者带来了不确定性,同时降低了微服务架构下系统的可攻击性,以此克服同构性微服务带来的安全隐患。谢记超等人[14]提出了一种基于节点异构性的部署方法,旨在进行冗余备份和重映射时保证节点的异构性。Wen 等人[15]提出了一个新的可靠的微服务编排框架,该框架采用遗传算法执行微服务组合,同时减少了用户安全要求和实际服务提供之间的差异。张淼等人[16]考虑到操作系统同构性可能引发的安全风险,利用异构操作系统提出了一种基于多样性的虚拟机安全部署策略,有效降低了攻击者的单位攻击收益,但由于只考虑了虚拟机的安全部署,并未考虑资源约束等问题。传统的部署与编排机制侧重于单个服务链的服务质量(quality of service,QoS)优化,而忽略了实例之间的共享和竞争。为了解决这个问题,Ding 等人[17]提出了一种基于列表调度的微服务选择算法(microservice service selection algorithm,MSS)。该算法采用工作流模型来对服务链进行描述,提出了基于子任务期限和紧迫性的两种服务选择策略,以完成微服务实例选择,构成服务链。Alleg 等人[2]提出了一种多样性和冗余联合机制,并提出利用混合整数线性程序(mixed-integer linear programming,MILP)为模型的服务功能链(service function chain,SFC)部署解决方案,旨在满足目标 SFC 可用性级别的同时,降低了由于多样性和冗余而导致的成本上升,避免了服务中断的同时,为备份实例分配了更少的资源。仝青等人[18]提出了一种自适应的时空多样性联合调度策略,该策略联合了时空多样性,结合粗粒度的入侵检测,克服了单一策略下防御代价高昂且效果不佳的缺陷,以较低代价提高了系统的防御能力,但并未讨论资源消耗带来的影响,也未将部署位置纳入考虑。

微服务架构下服务链的部署问题可以理解为在一定约束条件下的优化问题,该问题是一个 NP 难问题,在求解部署方案的算法选择上,研究人员通常选择遗传算法[15]、模拟退火算法[10]、列表搜索算法[17]等启发式算法,除此之外 Viterbi 算法[14,19]也被多次提及。

## 2 基于多样性的韧性微服务链动态部署模型

基于已有的研究可知,通过多样性技术能够有效提升软件系统的韧性抗毁能力,但在现有单体式架构下,实现软件系统的多样性面临较大的挑战。微服务技术的出现为解决这一问题提供了新的思路,本章主要研究如何在资源约束下建立一种多样化微服务的部署模型,以最大化系统的韧性抗毁能力。

### 2.1 基于多样性的韧性系统模型与故障场景

#### 2.1.1 基于多样性的韧性系统模型

本文所提出的部署策略是一种半在线的部署策略,系统每次能够处理一个批次的服务请求,每个服务请求对应着一个服务链,一条响应特定请求的服务链示例如图 1 所示。

图 1 的服务链示例由六个独立的微服务组件协调完成,服务链基本模型可以用图 1 所示的 DAG 图表示。其中,$D_i$ 表示该服务链上的第 $i$ 个微服务组件,int($i$,$j$)表示容器 $D_i$ 和 $D_j$ 之间的业务流链路开销。

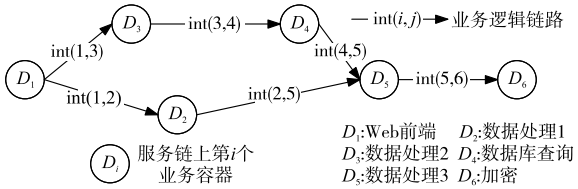


图 1 服务链示例

Fig. 1 Example of a service chain

图 2 所示是服务链上各容器与基础设施层各部署节点之间的映射关系,其中 $N_i$($i$ = 1,2,3,…)表示部署节点,节点既可能是某一台物理服务器,也可能是某个部署在物理机上的虚拟机,使用不同的 OS,包括 Debian、Ubuntu、CentOS 等。本文所研究的问题,就是如何根据请求将不同的服务链部署到具有多个部署节点的集群中,以满足用户和系统多样性需要。

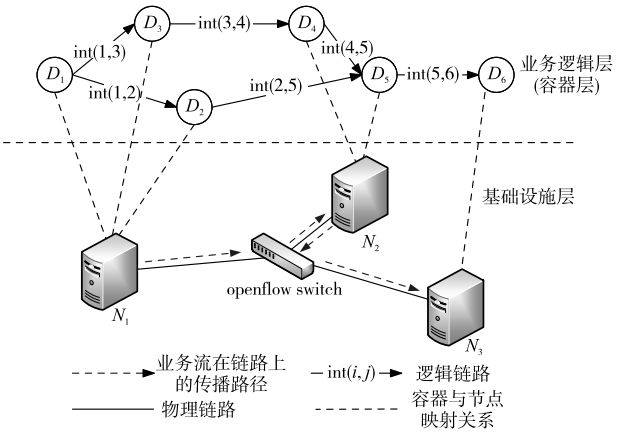


图 2 服务链部署示意图

Fig. 2 Service chain deployment diagram

#### 2.1.2 故障场景

1)共享漏洞问题 在使用容器构建软件系统时,由于大量使用相同的基础镜像,所以当镜像存在高危漏洞,并且当攻击者掌握了这一漏洞时,攻击者就能利用已知漏洞轻而易举地针对系统中的容器发起攻击,使之丧失服务能力。

2)节点故障问题 节点故障是一个不可避免的问题,当集群中某一个节点出现故障导致无法工作时,所有部署在该节点上的容器都将失去提供服务的能力,从而影响相关的服务请求完成情况。节点故障对服务请求的影响主要存在两个方面:a)主链上的容器失效,需要启用部署在其他节点上的冗余服务组件;b)不仅主链上的容器失效,由于其他冗余也部署在同一节点上,或是该服务并未部署冗余,从而导致服务请求的失败。本文在实验环节中也将考察在某一节点失效的情况下系统受影响的程度。

### 2.2 问题建模

#### 2.2.1 容器多样性模型

文献[16,20]根据生物多样性的度量方法,利用香农公式,提出了一种系统多样性的度量方法,对单个微服务而言,其数学公式可以表述为

$$D = -\sum_{i=1}^{d} p_i \log_2 p_i \tag{1}$$

其中:$D$ 表示该项微服务的多样性指标,$d$ 表示该微服务组件的实现版本数量,$p_i$ 表示第 $i$ 个版本的组件部署数量占部署总数的比例。因此根据多样性最大化的目标,求取的目标函数应该为 Max $D$,约束条件为

$$\sum_{i=1}^{d} p_i = 1 \tag{2}$$

$$0 \leqslant p_i \leqslant 1 \tag{3}$$

根据香农第一定理,多样性指标 $D$ 取极值当且仅当 $p_1 = p_2 = \cdots = p_d = \frac{1}{d}$ 时成立,最大值为 $\log_2 d$。

由此可见,在基于冗余和多样性部署时,对每个微服务而言,应当使部署的微服务组件数量在各版本间尽可能地平均分布,由此获得多样性的最大化。

### 2.2.2　韧性部署问题模型

微服务架构下软件系统在提供服务时,主要通过为用户请求分配并部署服务链实现,该部署主要解决服务链上每个微服务部署版本种类数量、各版本冗余部署数量以及部署位置三个问题。该问题可以理解为在一定约束条件下的优化问题,本文的优化目标为在满足软件系统可靠性的同时,使集群消耗的网络资源最小化。为对模型进行更好的解释说明,本文对即将使用的变量符号进行了解释说明,具体如表 1 所示。

表 1　部分符号说明

Tab. 1　Partial symbol interpretation

| 符号 | | 释义 |
|---|---|---|
| 服务链相关变量 | $Q_i$ | 第个 $i$ 请求($i \leqslant q$) |
| | $G_i$ | 第个 $i$ 请求对应的服务链 DAG 图 |
| | $V_i$ | $G_i$ 中的微服务组件(容器)类型集合 |
| | $\lvert V_i \rvert$ | $G_i$ 中的微服务组件(容器)类型数量 |
| | $E_i$ | $G_i$ 中的虚拟链路集合 |
| | $d_v$ | $G_i$ 中微服务 $v$ 的实现版本数量($v \in V_i$) |
| | $r_j^v$ | 微服务 $v$ 第 $j$ 个版本的冗余部署数量 |
| | $A$ | 用户请求可靠性要求 |
| | $P^v$ | $G_i$ 中微服务 $v$ 的故障概率 |
| | $p_j^v$ | 微服务 $v$ 第 $j$ 个版本实现的故障概率 |
| | $R_{v,j}^{CPU}$ | 部署微服务 $v$ 的第 $j$ 个版本实现所需的处理器资源 |
| | $R_{v,j}^{MEM}$ | 部署微服务 $v$ 的第 $j$ 个版本实现所需的内存资源 |
| | $R_{e,k}^{load}$ | 虚拟链路 $e$ 第 $k$ 条冗余的链路开销 |
| 可靠性相关变量资源相关变量 | $N$ | 部署节点集合 |
| | $L$ | 物理链路集合 |
| | $R_{N_x}^{CPU}$ | 部署节点 $N_x$ 所拥有的处理器资源($x \leqslant n$) |
| | $R_{N_x}^{MEM}$ | 部署节点 $N_x$ 所拥有的内存资源 |
| | $R_{L_y}^{load}$ | 物理链路 $L_y$ 的链路容量($y \leqslant l$) |
| | $H_{j,k}^v(N_x)$ | Bool 值,指示容器 v 是否在部署节点 $N_x$ 上 |
| | $H_k^e(L_y)$ | Bool 值,指示链路 $e$ 第 $k$ 条冗余是否由物理链路 $L_y$ 承载 |

从满足用户所需的服务可靠性出发,对于每个用户请求,都应该通过部署合适的多样性和冗余度,使其可靠性达到一个可接受的阈值 $A$。假设请求 $Q_i$ 对应的服务链 $G_i$ 中各微服务的故障概率为 $P^v$,其中 $v \in V_i$,那么服务链的可靠性约束可以描述为

$$\prod_{v \in V_i} (1 - P^v) \geqslant A \tag{4}$$

在式(4)的约束下,为防止某一微服务可靠性过低给服务链带来的可靠性瓶颈,因此本文对该约束条件进一步强化,进而要求每个微服务都达到一个最低可靠性阈值,如表达式(5)所示。

$$P^v \leqslant 1 - \sqrt[|V_i|]{A} \tag{5}$$

对于服务链中单个微服务 $v$ 而言,由于运用了多样性与冗余相结合的部署方式,各部署实例之间是并行的关系。在这种部署方式下,只有当部署的微服务 $v$ 全部版本的所有实例全部失效,微服务 $v$ 才会出现不可用的情况,因此其故障概率 $P^v$ 为

$$P^v = \prod_{j=1}^{d_v} (p_j^v)^{r_j^v} \tag{6}$$

式(6)说明了微服务的可靠性可以通过部署更多的实例得到提高,将表达式(6)代入约束(5)中,可以得到可靠性对于微服务部署的约束条件表达式:

$$1 - \prod_{j=1}^{d_v} (p_j^v)^{r_j^v} \leqslant \sqrt[|V_i|]{A} \tag{7}$$

除了可靠性约束外,服务链的部署还需要满足计算和网络资源约束,其约束条件可以表述为

$$\sum_{x=1}^{n} H_{j,k}^v(N_x) = 1 \quad \forall i \leqslant q \, \forall v \in V_i \tag{8}$$

$$\sum_{i=1}^{q} \sum_{v \in V_i} \sum_{j=1}^{d_v} \sum_{k=1}^{r_j^v} R_{v,j}^{CPU} H_{j,k}^v(N_x) \leqslant R_{N_x}^{CPU} \quad \forall x \tag{9}$$

$$\sum_{i=1}^{q} \sum_{v \in V_i} \sum_{j=1}^{d_v} \sum_{k=1}^{r_j^v} R_{v,j}^{MEM} H_{j,k}^v(N_x) \leqslant R_{N_x}^{MEM} \quad \forall x \tag{10}$$

$$\sum_{i=1}^{q} \sum_{e \in E_i} \sum_{k=1} R_{e,k}^{load} H_k^e(L_y) \leqslant R_{L_y}^{load} \quad \forall y \tag{11}$$

其中,式(8)表示服务链中的某一个容器只能被部署一次;式(9)(10)分别表示节点上部署的容器所需消耗的处理器、内存资源之和不超过节点所拥有的资源上限;式(11)为物理链路资源约束。本文优化目标为在确保多样性的基础上,尽可能地减少交互,在降低链路开销的同时,降低链路故障给服务带来的影响,目标函数如式(12)所示。

$$\text{Min} \sum_{y=1}^{l} \sum_{i=1}^{q} \sum_{e \in E_i} \sum_{k=1} R_{e,k}^{load} H_k^e(L_y) \tag{12}$$

## 2.3　模型分析

本文所提出的部署方法主要针对 2.1.2 节提出的两大问题,结合负载均衡方法,旨在提出一种基于多样性的微服务韧性部署方法,在降低共享漏洞和节点故障带来的损失、提高系统可用性的同时,尽可能降低链路资源开销,从而降低物理链路故障带来的损失,并使之具备更好的负载均衡效果。基于以上目标,本文所提出的部署方案需要考虑以下原则:

a)尽可能将同一服务链上的主执行容器部署在同一节点上。这一原则有助于降低依赖于单个节点的服务请求数量,同时有助于减少微服务组件在部署节点之间的交互,从而降低物理链路故障带来的损失。

b)多样性部署版本不能与主执行容器部署在同一节点上。这样做的优势在于能够避免由于节点损失导致的服务请求对应服务链上某一微服务功能彻底无法得到保证,从而导致服务请求的失败。同时将多样性部署版本部署到其他节点上,也有利于充分利用各部署节点上的碎片空间,并且将多样性部署版本部署到资源压力更轻的节点上,也有利于实现更好的负载均衡效果。

c)多样性均衡要求。多样性均衡主要是指,对于同一微服务而言,其部署各版本的容器数量应当尽可能保持均衡,其目的在于减轻由于共享漏洞问题带来的微服务提供服务能力损失过于严重,例如,当系统中大量的容器使用 Ubuntu 基础镜像构建容器,那么当该镜像出现高危漏洞并为攻击者所掌握时,系统将丧失大部分微服务的服务能力。

d)负载均衡。在进行部署时,除有类似于原则 a)所提出的特殊要求外,应尽可能选择资源压力较轻的节点进行部署。

e)资源约束。主要包括两个方面:节点资源约束和链路资源约束。节点资源约束是指单个节点上的可用处理器(CPU)、内存(MEM)资源有限,部署在节点上的容器所需资源之和不得超过节点资源总量;链路资源约束是指每条物理链路上承载的虚拟链路传输数据量之和不得超过链路容量。

# 3　算法描述

## 3.1　基于多样性的微服务编排算法

本文所使用的编排算法是考虑多样性部署需要所设计的,充分考虑了第 2.3 节中的多样性均衡要求。此外,由于越复杂的服务请求,对应服务链所需微服务越多,根据式(7)可以得知,其对于单个微服务的可用性要求越高。在多样性部署模式下,根据式(6),对于单个微服务而言部署的容器数量越多,其故障概率越低,可用性越高。因此,对于单个微服务而言,其所在的服务链越长,实现该微服务所需部署的容器数量就越多,这将导致大量的资源消耗。本文所给出的编排方案充分考虑了这一点,本方案优先考虑复杂服务请求,即优先处理更长的

服务链,同时在挑选微服务组件部署版本时,优先考虑故障率更低的实现版本。

本编排算法的伪代码如算法 1 所示,脚本的输入包括:服务请求信息、微服务信息、拓扑信息。其中,服务请求信息包括了请求序号、服务链长度、服务链的微服务组成信息,微服务信息包括了各微服务不同版本实现的资源消耗、故障率、当前部署数量,拓扑信息主要保存了各请求对应的服务链的图结构数据。

算法 1 Diversity-based microservices orchestration algorithm

输入:req_info: The service requests information; svc_info: Microservices information; topo_info: Topos information。

输出:Service chains orchestration information。

```
1 for i in range(len(req_info)):
2    req ← request whose service chain with max topo lenth
3    sv_chain ← req corresponding service chain
4    error_threshold ← use sv_chain calculate error_threshold with formula (7)
5    while ms in sv_chain:
6       choose the version which has min error rate
7       ms_error_rate ← min error rate
8       while ms_error_rate > error_threshold:
9          choose the version with has min error rate among the other versions
10         ms_error_rate ←ms_error_rate * min error rate
11      end
12 end
```

在算法 1 中,第 1 ~4 行表示优先处理复杂服务请求,并根据服务链长度计算单项微服务所需提供的可用性指标。第 5 ~7 行表示在确定了所需处理的服务请求及其对应的服务链后,逐个确定该链中所需部署的微服务,以及在该链中单个微服务所需提供的可用性程度。第 8 ~11 行表示对于单个微服务而言,优先选择故障概率低的实现版本,直到该项微服务整体可用性达到要求。在该算法中,假设一个批次有 $q$ 个服务请求,每个服务请求包含 $m$ 个微服务,每个微服务又存在 $v$ 种版本实现,那么该算法的时间复杂度可以表示为 $O(qmv)$。

### 3.2 最小负载部署算法 Load-Min

服务链部署问题可以理解为在一定约束条件下求最优解的问题,该问题求解过程是一个 NP 难问题[2]。由于网络规模庞大,通过线性规划寻找最优解决方案复杂度高、耗时长,由于用户请求对即时性有严格的要求,而线性规划无法在有效时间内得到有效解,所以此类问题通常使用启发式算法。本文提出了一种最小负载部署算法(Load-Min),该算法是一种启发式的半在线部署算法,该算法按时间窗口分批次处理服务请求,每批次服务请求数量与当前用户需求与系统状态相关。

本文在算法 2 中给出了部署算法的伪代码,其输入包括:服务请求信息、微服务信息、拓扑信息、编排信息、节点信息,其中前三项在算法 1 中已经进行过说明,在此不再赘述。第四项编排信息是由算法 1 的执行结果得到的,关于每个服务链中各微服务具体要部署哪些容器的信息。第五项节点信息保持着各节点实时的状态信息,包括资源负载等。

算法 2 Load-Min deployment algorithm

输入:req_info: The service requests information; svc_info: Microservices information; topo_info: Topos information; orche_info: Orchestration information; node_info: Node information。

输出:Service chains deployment information。

```
1 for i in range(len(req_info)):
2    req ← request whose service chain with max topo lenth
3    sv_chain ← req corresponding service chain
4    node_main ← choose the node with min resource utilization rate
5    while ms in sv_chain:
6       while resource is not enough to deploy the first docker of ms:
7          node_main ← choose the node with min resource utilization rate among the other nodes
8       end
9       deploy the first docker on the node chosen
10      update node_info
11      // deploy the other dockers
12      for other dockers of ms:
13         node_vice ← choose the node with min resource utilization rate among the other nodes
14         while resource is not enough to deploy the docker:
15            node_vice ← choose the node with min resource utilization rate among the other nodes
16         end
17         deploy the docker on the node chosen
18         update node_info
19 end
```

在算法 2 中,第 1 ~4 行和算法 1 一样,表示优先处理复杂服务请求,并根据各节点负载状态获取当前负载最轻的节点作为当前请求服务链的主部署节点,第 5 ~10 行表示该服务链上所有微服务的主执行容器都部署在该节点上,直到资源无法承受,再考虑部署到下一个节点上,这样做的优势是尽可能将同一服务请求的主执行容器集中在单一节点上,既能降低物理链路负载,又能减少对单一节点产生依赖的服务请求数量。第 12 ~19 行表示多样性部署版本的部署,这类容器除了需要注意不与服务链中本微服务下其他微服务部署在同一节点上,其他主要考虑负载均衡效果,每次部署时都是种选择资源负载最低的节点进行部署。在该算法中,假设一个批次有 $q$ 个服务请求,每个服务请求包含 $m$ 个微服务,每个微服务存在 $v$ 种版本实现,集群中有 $n$ 个部署节点,那么该算法的时间复杂度可以表示为 $O(qm(1\times n+(v-1)\times n\times n/2)$。

## 4 性能评估

本文首先对模型所描述的系统进行了简单实现,并对其进行部分性能验证。为更好地评估模型性能,本文针对更复杂条件下的情况进行了仿真。

### 4.1 基于多样性的韧性部署原型系统实现

#### 4.1.1 基于多样性的微服务架构软件系统原型

在基于软件定义网络(software define network, SDN)实现的软件系统网络中,可以将编排器集成在 SDN 控制器中,使之成为整个软件系统的控制中心。如图 3 所示,SDN 控制器能够根据用户请求和资源约束定制部署策略,并将部署策略下发至集群各服务器中执行。在本文所设计的原型系统中,根据是否启用虚拟机,集群中的服务器分为两种:直接作为服务部署节点部署微服务组件(即业务容器)以及在服务器上启动虚拟机作为服务容器部署节点,如图 3 中的服务器 A。

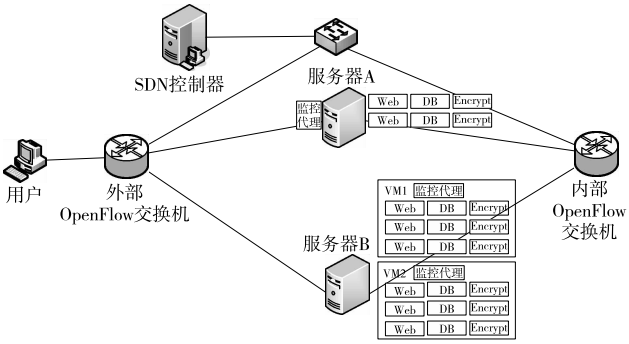


图 3 原型系统示意图

Fig. 3 Prototype system diagram

本文使用 Docker 容器技术,Docker 提供的虚拟化技术使得研究能够忽略底层平台的差异性,因此本文统一将诸如服务器 A、VM1、VM2 这样直接部署容器的主机视为部署节点。对于在节点上部署的业务容器而言,同属于同一微服务的容器有多个版本的实现,即使用不同容器镜像构建的具有相同业务功能的容器模板。例如,对于实现 Web 前端功能的容器而言,容器构建使用的基础镜像可以使用 Ubuntu、Debian、CentOS 等,不同版本的实现会带来不同的资源开销。在本文所描述的软件系

统中,系统业务被解耦成多个微服务,包括 Web 前端、数据库查询、加密组件等,不同微服务之间相互协调从而构成服务链。

#### 4.1.2　实验分析

为实现本文提出的基于多样性的微服务韧性部署方法,本文设计了第 3 章所述的原型系统,该系统包括一个 SDN 控制器(集成微服务编排器),两台服务器(共三个部署节点),一台二层交换机,两台 Openflow 交换机,一台 PC 作为用户。

为了验证策略实施的可行性和算法的有效性,本文在进行仿真实验之前,预先在原型系统"运力查询系统"上进行了验证实验。原型系统包括三种微服务:Web、Database 和 Encrypt,每类微服务都分别基于 Ubuntu、CentOS 和 Debian 进行了三种实现,九种容器的资源消耗情况如表 2 所示。服务请求包括两种实现:a)使用三种微服务进行响应的加密查询;b)使用两种微服务进行响应的普通查询(不使用 Encrypt 服务),随机使用两种服务请求进行 10 次操作进行验证。

表 2　容器资源消耗情况

Tab. 2　Resource consumption of the containers

| 容器 | 占用磁盘空间/MB | CPU 占用 | 内存占用/KB | 容器 | 占用磁盘空间/MB | CPU 占用 | 内存占用/KB |
|---|---|---|---|---|---|---|---|
| Web_Ubuntu | 332 | 0.02 | 3 116 | Database_Debian | 583 | 0.13 | 2 562 |
| Web_CentOS | 477 | 0.03 | 3 544 | Encrypt_Ubuntu | 177 | 0.01 | 1 543 |
| Web_Debian | 333 | 0.02 | 3 256 | Encrypt_CentOS | 418 | 0.02 | 2 013 |
| Database_Ubuntu | 574 | 0.12 | 2 388 | Encrypt_Debian | 163 | 0.01 | 1 569 |
| Database_CentOS | 640 | 0.20 | 2 655 | | | | |

如图 4、5 所示,整体上看,Load-Min 算法具有良好的负载均衡能力,这种均衡不仅体现在节点压力上,同时部署的各微服务版本间也存在较好的均衡,这有利于在面对攻击者针对某一已泄漏容器漏洞发起攻击时,尽可能降低损失,避免系统丧失提供服务能力。

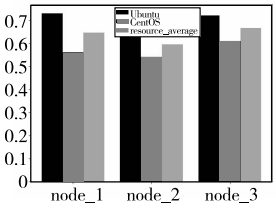


图 4　Load-Min 算法节点资源占用情况

Fig. 4　Node resource usage when using the Load-Min algorithm

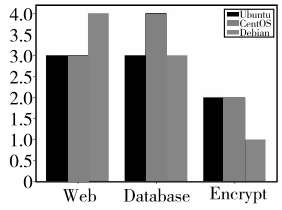


图 5　不同微服务版本分布情况

Fig. 5　Distribution of different microservices versions

### 4.2　基于多样性的韧性部署仿真实验

#### 4.2.1　实验设置

a)平台。本文仿真实验在 DELL T7920 工作站上执行,其处理器为 Intel® Xeon Gold 6248R CPU@3.00 GHz×96(双处理器),内存大小为 512 GB,使用 Ubuntu Server 操作系统。为了验证策略性能,本文使用 Python 3.7 进行仿真。

b)数据。本文所涉及的主要仿真参数如表 3 所示。

表 3　仿真参数

Tab. 3　Simulation parameters

| 参数 | 取值 |
|---|---|
| 部署节点数量 | 8 |
| 节点 CPU 资源 | [1,2,3](核) |
| 节点内存资源 | [1 024,2 048,3 096](M) |
| 请求服务链长度 | 服从[1,9]上的平均分布 |
| Docker CPU 占用率 | 服从[0.01,0.25](核)上的平均分布 |
| Docker MEM 占用率 | 服从[1,32](M)上的平均分布 |
| Docker 故障率 | 服从[0.01,0.10]上的平均分布 |
| 虚拟链路负载 | 服从[0.1,10] 上的平均分布 |

c)方法。将本文所使用的部署策略与六种算法进行对比,其中包括四种经典算法与两种新近算法。

其中四种经典算法包括:

(a)贪婪负载均衡算法(LB-greedy):该算法仅以追求负载均衡为目标,无论在确定实例版本还是节点负载上,每一次选择都选择使用最少的版本和负载最低的节点来承载实例。

(b)轮询算法(Round-Robin):该算法维护一个顺序列表,在每次选择实例部署版本和部署节点时都采用轮询方式,这一方式有利于负载均衡的实现,是一种典型的负载均衡方法。

(c)随机部署算法(Random):该算法在每次选择实例部署版本和部署节点时都采用随机选择的方式,这一方式有利于负载均衡的实现,是一种典型的负载均衡方法。

(d)集中部署算法(Concentrate):该算法在选择部署实例版本时与 Load-Min 类似,但在每次选择部署节点时都尽可能将同属于一个服务链的实例部署在同一节点上,这一方式有利于减少节点间的交互,减少链路负载的同时降低对链路的依赖。

两种新近方案包括:

混合整数线性程序(MILP)[2]:在满足目标服务链可用性级别的同时,降低由于多样性和冗余而导致的资源成本,有利于在避免服务中断的同时,为备份实例分配更少的资源。

基于列表调度的微服务选择算法(MSS)[17]:这是一种基于子任务期限和紧迫性的服务选择策略,通过这种方式完成微服务实例选择,从而构成服务链。

将本文所提出的 Load-Min 算法分别在负载均衡、共享漏洞、节点故障三个方面与上述六种经典算法进行对比考察。

d)指标。主要包括两个方面四项指标:资源效能方面,主要包括负载均衡和链路负载两点,负载均衡考察各节点之间资源利用率的标准差,标准差越小说明各节点之间的负载越均衡。安全性能方面,主要考察在共享漏洞问题和节点故障问题中,各部署策略下服务请求的完成情况,包括受影响请求数和失败服务请求数两个指标。受影响请求是指服务链主链上的主执行容器遭到破坏无法继续使用,从而启用多样性冗余的请求,该请求仍能完成,只不过性能会有略微影响。失败服务请求是指由于故障等原因导致该服务链上某一微服务的实现被全部摧毁,无法继续提供此项服务,因此服务请求无法完成。

#### 4.2.2　资源效能评估

1)负载均衡　本节主要考察不同的部署方法下的负载均衡表现,如图 6 所示,其中(a)~(e)表示不同算法实现下,各节点资源的负载情况,图中 cpu_usage_percentage 表示节点 CPU 利用率,mem_usage_percentage 表示节点内存利用率,resource_average 为上述两项的加权和。

$$\begin{cases} resource\_average = \alpha \cdot cpu\_usage\_percentage + \\ \quad \beta \cdot mem\_usage\_percentage \\ \alpha + \beta = 1 \,\&\, \alpha \in (0,1) \,\&\, \beta \in (0,1) \end{cases} \tag{13}$$

其中:$(\alpha,\beta)$ 为权重参数,表示对该项的重视程度。例如,在 CPU 密集型任务中,显然 CPU 的状态是更值得关注的对象,那么 $\alpha$ 的取值将远大于 $\beta$,由于本实验中整体上来看 CPU 与内存负载压力相差不大,因此取 $\alpha=\beta=0.5$。图 6 展示了不同算法下各节点 resource_average 的标准差,可见负载均衡表现最好的是 LB-greedy 算法,稍好于 Load-Min 算法,但差距并不明显,这是由于 LB-greedy 算法单纯只追求负载均衡的结果,而 Load-Min 算法则需要兼顾主链集中部署的原则。除 LB-greedy 外的三种经典算法表现则与前两者存在相当大的差距,此外,可见两种新近算法也将负载均衡要素考虑在内。

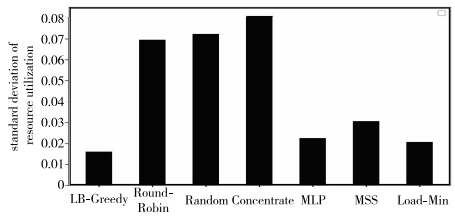


图 6　不同部署策略下节点资源占用情况

Fig. 6　Node resource usage under different deployment strategies

2)链路负载 如表 4 所示,由于 LB-Greedy 算法、round-Robin 算法、random 算法在每次部署时更倾向于将同一服务链的容器部署到不同的节点上,因此利用这三种算法部署得到的链路负载远远大于另外两种新算法。可见虽然 LB-Greedy 算法获得了最好的负载均衡效果,但却带来了巨大的链路负载压力。

表 4 各部署算法下的链路负载

Tab. 4 Link load under different deployment algorithm

| 部署算法 | 链路负载 | 部署算法 | 链路负载 | 部署算法 | 链路负载 |
|---|---|---|---|---|---|
| LB-Greedy | 367.722 4 | Concentrate | 95.267 4 | MSS | 102.264 8 |
| Round-Robin | 394.490 8 | MILP | 186.265 9 | Load-Min | 7.212 6 |
| Random | 357.088 3 | | | | |

Load-Min 算法在链路负载方面的表现是所有算法中最好的,甚至远小于 Concentrate。Concentrate 本身是一种集中式的部署方法,这种方式是有利于降低链路负载的,但是由于该算法必须要填满一个节点后才能在下一个节点上进行部署,这种方式将导致大量的请求服务链部署在前后相邻的两个节点上,而 Load-Min 则灵活选择资源压力最小的节点部署整个主链,使尽可能多的服务链主链部署在单一节点上,从而带来了相较于 Concentrate 算法实现更小的链路负载。此外,两种新近策略中,MSS 的链路负载显然小于其余多数方案,甚至接近 Concentrate 的结果,原因是该算法需要强调响应时间,在这一目标下服务链将尽可能被约束在同一节点下,以此达到缩减响应时间的目标。而 MILP 的负载同样相对较低的原因在于在该模型中,链路资源也是需要降低的资源成本之一。

### 4.2.3 安全性能评估

1) 单镜像漏洞泄露下对服务请求的影响

共享漏洞问题本质上是已经部署了服务的容器中,当某一类容器集体失效时,使得部分微服务失去提供服务的能力,从而导致服务链的不完整,最终使得服务请求失败。共享漏洞问题主要与该微服务所部署的容器多样性相关,由于所有容器在部署之前就已经通过编排得到了所要部署的微服务组件容器版本数量与部署数目,所以本小节主要考察在不同多样性下用户服务请求的受影响及完成情况。如图 7 所示,随着多样性的增减,受影响的服务请求数不断减少,而在图 8 中则显示,当多样性水平(即每个微服务的实现版本数)大于等于 2 后,彻底失去服务能力而导致失败的请求数就快速下降到了较低水平。

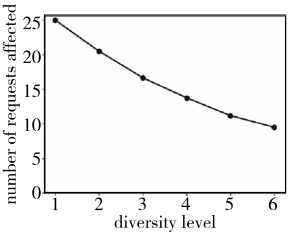


图 7 单镜像漏洞泄漏下受影响服务请求数

Fig. 7 The number of affected service requests under a single-mirror vulnerability leak

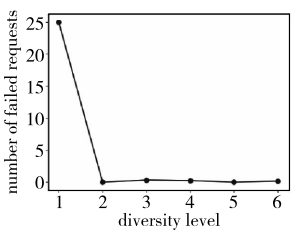


图 8 单镜像漏洞泄漏下无法完成的服务请求数

Fig. 8 The number of failed service requests under a single-mirror vulnerability leak

2)单节点故障下对服务请求的影响

当集群中某一个部署节点发生故障时,部署在该节点上的容器就会全部失去其提供服务的能力,从而对服务请求产生影响。为了更准确地评估这一故障带来的损失,分别在每一批请求数量分别为{5,10,15,20,25}的情况下,针对每一个节点都可能发生的故障情况进行了反复实验,通过多次实验取均值的方式获取了在不同部署模式下,随着服务请求数量变化,而产生的服务请求受影响情况。根据图 9 能够获知,随着每一批次处理的服务请求数量不断增加,因单节点故障而受影响的服务请求数量逐步上升,其中表现最好的是 Concentrate 算法和 Load-Min 算法,这是因为这两种算法尽可能将同一服务链的主链放置到同一节点上的缘故,这导致了节点上的大量资源被少数几个服务请求占据,因此在单节点故障时受影响最小。而另外三种经典算法则恰恰相反,另外三种都趋向于将容器分开部署,这样部署的结果是单个节点的资源被大量服务请求共享,使得在节点故障时有大量的服务请求受到影响。两种新近算法由于链路资源等原因考虑了将服务链上的实例集中部署,所以相对于上述三种经典算法有更好的表现。

图 10 则展示了七种不同算法的部署结果在承受单节点损失下无法完成的服务请求数(图中 Round-Robin 与 Load-Min 重合),在图 9 中有着较好表现的 Concentrate 算法在这种情况下则表现最为糟糕。由于 Concentrate 算法在集中部署时,不仅将同一请求的主链部署在了同一节点上,其多样性版本也同样部署在了同一节点上,这导致了一旦节点发生故障无法继续工作,所有将全部实例都部署在该节点上的微服务失去服务能力,从而导致服务请求失败。Load-Min 算法则巧妙地避开了这一尴尬结果,Load-Min 算法只把主链部署在同一节点上,并且在部署其他版本的容器时,尤其注意避开已部署了同一微服务的节点,并尽可能分散部署,在分散这一点上,Load-Min 和 Round-Robin 是一致的,因此两者都在单节点故障下具有最小的损失。

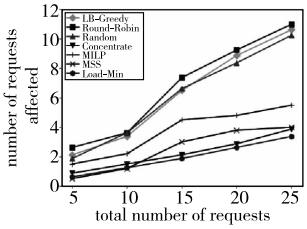


图 9 单节点故障下受影响服务请求数

Fig. 9 The number of service requests affected under a single-node failure

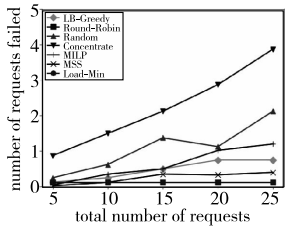


图 10 单节点损失下无法完成的服务请求数

Fig. 10 Number of service requests that cannot be completed under single-node loss

## 5 结束语

本文针对微服务架构下如何基于多样性对异构容器进行部署提出了一种最小负载部署算法(Load-Min)。该方案结合了集中式部署与负载均衡算法的优势,既保留了集中式部署链路资源消耗低的优势,又兼顾了负载均衡,降低了对单节点的依赖。结合应用多样性条件下的编排算法,本文对 Load-Min 算法和其他六种算法进行了比较,实验结果显示 Load-Min 部署算法相较其他经典算法,显得更加高效、安全和可靠。

本文提出的 Load-Min 算法是一种半在线部署方法,这种方法需要按时隙分批次对服务请求进行处理,这种处理方式每批次能处理的服务请求有限,并且这种部署方式可能增大服务时延。针对上述问题,下一步的研究工作将致力于研究完全在线的部署方案,甚至可以利用机器学习的手段对服务持续时间进行预测,从而更精确地对实时响应的服务请求作出更加合理的部署,进一步提高资源利用率,缩短响应时间。

**参考文献:**

[1] Gesvindr D, Davidek J, Buhnova B. Design of scalable and resilient applications using microservice architecture in paas cloud[C]//Proc of International Conference on Software Technologies. 2019: 619-630.

[2] Alleg A, Ahmed T, Mosbah M, *et al*. Joint diversity and redundancy for resilient service chain provisioning[J]. IEEE Journal on Selected Areas in Communications, 2020, 38(7): 1490-1504.

[3] Esposito C, Castiglione A, Choo K-K R. Challenges in delivering software in the cloud as microservices[J]. IEEE Cloud Computing, 2016, 3(5): 10-14.

[4] Douglis F, Nieh J. Microservices and containers[J]. IEEE Internet Computing, 2019, 23(6): 5-6.

[5] Vayghan L A, Saied M A, Toeroe M, *et al*. Microservice based architecture: Towards high-availability for stateful applications with kubernetes[C]//Proc of IEEE International Conference on Software Quality, Reliability and Security. Piscataway, NJ: IEEE Press, 2019: 176-185.

(下转第 页)

机室内定位算法[J]. 计算机应用研究,2016,33(6):1849-1852. (Jing Shoucai, Hui Fei, Ma Xupan, *et al*. Smartphone indoor location algorithm based on RSSI and motion sensor fusion[J]. Application Research of Computers,2016,33(6):1849-1852.)

[4] Mirowski P, Milioris D, Whiting P, *et al*. Probabilistic radio-frequency fingerprinting and localization on the run[J]. Bell Labs Technical Journal,2014,18(4):111-133.

[5] 刘国栋. 基于 UWB 的室内定位技术研究[D]. 南京:南京邮电大学,2014. (Liu Guodong. Indoor location technology research based on UWB[D]. Nanjing:Nanjing University of Posts & Telecommunications,2014.)

[6] Kang W,Han Y. SmartPDR: smartphone-based pedestrian dead reckoning for indoor localization[J]. IEEE Sensors Journal,2015,15(5):2906-2916.

[7] Davidson P, Piche R. A survey of selected indoor positioning methods for smartphones[J]. IEEE Communications Surveys & Tutorials,2017,19(2):1347-1370.

[8] Harle R. A survey of indoor inertial positioning systems for pedestrians[J]. IEEE Communications Surveys & Tutorials,2013,15(3):1281-1293.

[9] Skog I, Handel P, Nilsson J O, *et al*. Zero-velocity detection:an algorithm evaluation[J]. IEEE Trans on Biomedical Engineering,2010,57(11):2657-2666.

[10] 张超翔,陈璟,郑晨辉. 基于峰值检测的自适应时间窗口计步算法[J]. 电子测量与仪器学报,2021,35(4):195-203. (Zhang Chaoxiang, Chen Jing, Zheng Chenhui. Adaptive time window step counting algorithm based on peak detection[J]. Journal of Electronic Measurement and Instrumentation,2021,35(4):195-203.)

[11] Xie Hongwei, Gu Tao, Tao Xianping, *et al*. MaLoc: a practical magnetic fingerprinting approach to indoor localization using smartphones[C]//Proc of ACM International Joint Conference on Pervasive and Ubiquitous Computing. New York:ACM Press,2014:243-253.

[12] Li Xin, Wang Jian, Liu Chunyan. A Bluetooth/PDR integration algorithm for an indoor positioning system[J]. Sensors,2015,15(10):24862-24885.

[13] 亢璐. 基于 Wi-Fi 指纹与 PDR 融合的室内定位算法研究[D]. 武汉:华中科技大学,2019. (Kang Lu. Research on indoor positioning based on fusion of Wi-Fi fingerprint and PDR[D]. Wuhan:Huazhong University of Science & Technology,2019.)

[14] Rudi B, Klaffenbck M A, Pichler-Scheder M, *et al*. Geometry-aided BLE-based smartphone positioning for indoor location-based services[C]//Proc of IEEE MTT-S International Conference on Microwaves for Intelligent Mobility. Piscataway,NJ:IEEE Press,2020.

[15] Mouhammad C S, Allam A, Abdel-Raouf M, *et al*. BLE indoor localization based on improved RSSI and trilateration[C]//Proc of the 7th International Japan-Africa Conference on Electronics, Communications, and Computations. Piscataway,NJ:IEEE Press,2019:17-21.

[16] Liao J K, Chiang K W, Zhou Zhiming. The performance analysis of smartphone-based pedestrian dead reckoning and wireless locating technology for indoor navigation application[J]. Inventions,2016,1(4):25.

[17] 史卓瑛. 面向空旷场景基于移动设备的室内定位与导航系统[D]. 杭州:浙江大学,2018. (Shi Zhuoying. Indoor localization and navigation syatem for mobile devices in spacious environment[D]. Hangzhou:Zhejiang University,2018.)

[18] Wang Mei, Duan Nan, Zou Zhou, *et al*. Indoor localization on smartphone using PDR and sparse deployed BLE beacons in large open area[C]//Proc of International Wireless Communications and Mobile Computing Conference. Piscataway,NJ:IEEE Press,2021:141-148.

[19] Hölzke F, Wolff J, Haubelt C. Improving pedestrian dead reckoning using likely paths and backtracking for mobile devices[C]//Proc of IEEE International Conference on Pervasive Computing and Communications. Piscataway,NJ:IEEE Press,2019:273-278.

(上接第 页)

[6] Wang Sheng, Ding Zzhijun, Jiang Changjun. Elastic scheduling for microservice applications in clouds[J]. IEEE Trans on Parallel and Distributed Systems,2021,32(1):98-115.

[7] Combe T, Martin A, Pietro R D. To docker or not to docker: a security perspective[J]. IEEE Cloud Computing,2016,3(5):54-62.

[8] Mijumbi R, Serrat J, Gorricho J-L, *et al*. Network function virtualization: state-of-the-art and research challenges[J]. IEEE Communications Surveys & Tutorials,2017,18(1):236-262.

[9] Wan Xili, Guan Xinjie, Wang Tianjing, *et al*. Application deployment using microservice and docker containers: framework and optimization[J]. Journal of Network and Computer Applications,2018,119:97-109.

[10] 谷允捷,胡宇翔,谢记超. 基于重叠网络结构的服务功能链时空优化编排策略[J]. 电子与信息学报,2019,41(11):2675-2683. (Gu Yunjie, Hu Yuxiang, Xie Jichao. A spatial and temporal optimal method of service function chain orchestration based on overlay network structure[J]. Journal of Electronics & Information Technology,2019,41(11):2675-2683.)

[11] Han Jungsu, Hong Yujin, Kim Jongwon. Refining microservices placement employing workload profiling over multiple kubernetes clusters[J]. IEEE Access,2020,8:192543-192556.

[12] Joseph C T, Chandrasekaran K. Intma: dynamic interaction-aware resource allocation for containerized microservices in cloud environments[J]. Journal of Systems Architecture,2020,111:101785-101799.

[13] Torkura K A, Sukmana M I H, Kayem A V D M. A cyber risk based moving target defense mechanism for microservice architectures[C]//Proc of IEEE International Conference Ubiquitous Computing and Communication. Piscataway,NJ:IEEE Press,2018:932-939.

[14] 谢记超,伊鹏,张震,等. 基于异构备份与重映射的服务功能链部署方案[J]. 网络与信息安全学报,2018,4(6):23-35. (Xie Jichao, Yi Peng, Zhang Zhen, *et al*. Service function chain deployment scheme based on heterogeneous backup and remapping[J]. Chinese Journal of Network and Information Security,2018,4(6):23-35.)

[15] Wen Zhenyu, Lin Tao, Yang Renyu, *et al*. GA-Par:dependable microservice orchestration framework for geo-distributed clouds[J]. IEEE Trans on Parallel and Distributed Systems,2019,31(1):129-143.

[16] 张森,季新生,艾健健,等. 基于操作系统多样性的虚拟机安全部署策略[J]. 网络与信息安全学报,2017,3(10):35-43. (Zhang Miao, Ji Xinsheng, Ai Jianjian, *et al*. Secure deployment strategy of virtual machines based on operating system diversity[J]. Chinese Journal of Network and Information Security,2017,3(10):35-43.)

[17] Ding Zhijun, Wang Sheng, Pan Meiqin. Qos-constrained service selection for networked microservices[J]. IEEE Access, 2020, 8:39285-39299.

[18] 仝青,郭云飞,霍树民,等. 自适应的时空多样性联合调度策略设计[J]. 通信学报,2021,42(7):12-24. (Tong Qing, Guo Yunfei, Huo Shumin, *et al*. Design of self-adaptive spatio-temporal diversity joint scheduling strategy[J]. Journal on Communications,2021,42(7):12-24.)

[19] 刘彩霞,卢干强,汤红波,等. 一种基于 Viterbi 算法的虚拟网络功能自适应部署方法[J]. 电子与信息学报,2016,38(11):2922-2930. (Liu Caixia, Lu Ganqiang, Tang Hongbo, *et al*. Adaptive deployment method for virtualized network function based on Viterbi algorithm[J]. Journal of Electronics & Information Technology,2016,38(11):2922-2930.)

[20] Tong Qing, Guo Yunfei, Hu Hongchao, *et al*. A diversity metric based study on the correlation between diversity and security[J]. IEICE Trans on Information and Systems,2019,E102D(10):1993-2003.