软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(5):1341−1359 [doi: 10.13328/j.cnki.jos.006236]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

多版本共存的微服务系统自适应演化方法*

贺 祥, 刘 磊, 涂志莹, 徐晓飞

(哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

通讯作者: 徐晓飞, E-mail: xiaofei@hit.edu.cn

摘 要: 微服务设计模式通过将应用程序拆分成多个相互独立的微服务,实现了各个微服务之间的相互解耦,允许各个微服务能够独立地进行迭代开发、部署,从而对用户需求变化以及 DevOps 流程中部署需求作出快速响应.每个微服务的独立迭代升级导致了系统中可能出现多版本共存现象,不同服务的不同版本之间的依赖关系变得更加复杂.如何在这种场景下适应用户不断变化的需求以及开发者敏捷 DevOps 流程中部署需求,是当前面临的一个挑战.为解决这一问题,提出了微服务依赖模型来刻画不同服务的不同版本之间复杂的依赖关系,设计了基于贪婪的优化算法来找到最优的微服务系统演化方案,以满足用户需求变化和敏捷 DevOps 流程中部署需求,并实现了面向演化的微服务编程框架(MF4MS)和微服务系统自适应架构(MI4MS),可支持演化方案的自动执行,实现微服务系统运行时的自适应演化.实验结果表明:在有着复杂依赖的微服务系统中,该方法在服务失效时长、服务可用性、开发者 DevOps 代价等指标上有很好的表现,可有效支持微服务系统自适应演化,以应对用户需求变化和敏捷 DevOps.

关键词: 微服务系统;多版本共存;版本依赖;自适应;用户需求变化;DevOps

中图法分类号: TP311

中文引用格式: 贺祥,刘磊,涂志莹,徐晓飞.多版本共存的微服务系统自适应演化方法.软件学报,2021,32(5):1341−1359. http://www.jos.org.cn/1000-9825/6236.htm

英文引用格式: He X, Liu L, Tu ZY, Xu XF. Self-adaptative evoluationary method of multi-version coexisting microservice systems. Ruan Jian Xue Bao/Journal of Software, 2021,32(5):1341−1359 (in Chinese). http://www.jos.org.cn/1000-9825/6236.htm

Self-adaptative Evoluationary Method of Multi-version Coexisting Microservice Systems

HE Xiang, LIU Lei, TU Zhi-Ying, XU Xiao-Fei

(School of Computer Science and Technology, Harbin Institute of Technology, Harbin 150001, China)

Abstract: A microservice-based system is composed of a set of microservices that are developed and deployed independently for agile DevOps. Intensive and iterative adaptations/upgrades of microservices are essential for such systems to adapt to user requirement changes and DevOps, and as a consequence, result in the phenomenon of “multi-version microservice coexistence” in a system. Besides traditional API-based functional dependencies between different microservices, there appear complicated dependencies between different versions of difference microservices, which dramatically deteriorate the maintainability of microservice systems, especially when systems evolve to adapt to user requirement changes and DevOps. To meet this challenge, a version dependency model is proposed for describing the complex dependencies between different versions of microservices, and greedy-based optimization algorithms are developed for generating an optimal evolution plan according to changes in user requirements and DevOps at different scenarios. A programming framework (MF4MS) and cloud-edge based infrastructure (MI4MS) are implemented to facilitate microservice systems to automatically

* 基金项目: 国家重点研发计划(2018YFB1402500); 国家自然科学基金(61832014, 61772155, 61832004, 61802089)

Foundation item: National Key Research and Development Program of China (2018YFB1402500); National Natural Science Foundation of China (61832014, 61772155, 61832004, 61802089)

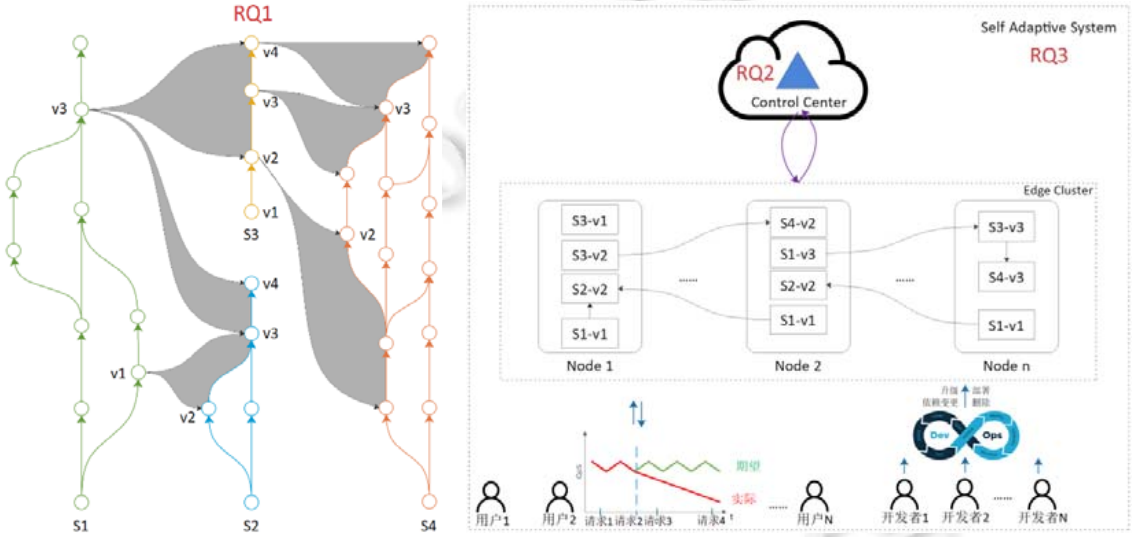
本文由“面向持续软件工程的微服务架构技术”专题特约编辑张贺教授、王忠杰教授、陈连平研究员和彭鑫教授推荐.

收稿时间: 2020-09-21; 修改时间: 2020-10-26; 采用时间: 2020-12-15; jos 在线出版时间: 2021-02-07

execute the evolution plan. Experiments show that the proposed approach performs well on service down time, service availability, and the developers' cost of DevOps coping with self-adaptation in the situation where complicated version dependencies exist.

Key words: microservice systems; multi-version coexistence; version dependency; self adaptation; user requirement changes; DevOps

在当今的一些诸如智慧城市(smart city)[1]等较为复杂的计算场景中,由于容器技术和微服务架构模式[2]在持续交付、敏捷开发以及 DevOps 方面的优势[3],随着计算场景中的业务逻辑变得越来越复杂,容器技术和微服务架构模式受到越来越多的关注.然而,微服务的独立开发和部署,导致它们之间可能存在复杂的依赖关系.微服务通过 API 相互通信,它们之间存在的调用依赖关系可表示为服务依赖关系图(SDG)[4].将某个微服务升级到新版本后,它的调用依赖关系可能会变化,依赖于该微服务的其他微服务可能会由于不能向前兼容的升级而无法正常调用.考虑到某些用户会请求特定的旧版本,微服务开发人员无法更新系统中已部署的所有微服务实例.因此,同一个微服务会有多个不同版本在系统中同时部署,这称为多版本共存(multi-version coexistence).在多版本共存的微服务系统中,不同微服务的不同版本之间有着不同的依赖关系,这种复杂的依赖关系称为版本依赖(version dependency),如图 1(a)所示,服务 *S*1 在版本 *v*1 时,依赖服务 *S*2 的 *v*2,*v*3 版本.而当服务 *S*1 升级至 *v*3 版本后,依赖变为服务 *S*2 的 *v*3,*v*4 版本以及服务 *S*3 的 *v*2,*v*3,*v*4 版本.



(a) 版本依赖 (b) 系统架构概观

Fig.1 Introduction for multi-version coexisting microservice system

图 1 多版本共存的微服务系统介绍

这种复杂的微服务系统面临着频繁的演化需求.

- 一方面,此类系统往往存在着大量的用户和服务,且用户需求处于时刻变化中.而系统无法对变化的用户需求作出及时响应,导致系统的服务质量(QoS)下降.由于多版本共存导致版本依赖性的复杂性增加,在边缘计算的微服务场景中,如何在复杂版本依赖的情况下让服务系统对用户需求变化作出及时响应,成为了一个新的挑战.为此,需要形成一个针对需求动态变化的演化方案.
- 另一方面,开发者为了完善系统,不断在 DevOps 流程中提出需求,如部署新服务、管理现有实例等操作.服务之间复杂的版本依赖,让一项 DevOps 流程中部署需求的实现不再是原子的,而是可能影响到其他微服务的其他版本的 DevOps 流程中部署相关的操作,增加了开发者运维微服务系统的负担.

综合起来,考虑到系统中大量的用户、服务以及依赖关系,在云边缘环境中的微服务系统应能够在满足版本依赖的情况下:(1) 针对用户需求变化进行自适应演化以保持 QoS 稳定[5];(2) 正确和自适应地满足开发者的

DevOps流程中部署需求.该问题如图1(b)所示.考虑到DevOps是一套完整的快速交付流程,本文仅考虑DevOps流程中和服务实例部署相关的操作.

在本文中,我们考虑了有关云边缘环境中多版本共存微服务系统自适应演化的3个研究问题(RQ).

RQ1. 如何对微服务之间的版本依赖进行建模?由于每个微服务的版本树都是独立的,版本依赖关系会随时间变化,因此对于服务系统而言,在运行时对版本依赖关系进行建模至关重要.此外,现有描述微服务之间调用依赖关系的方法如 SDG 无法应对微服务的快速迭代开发.当微服务发生兼容升级时,调用该微服务的其他微服务应动态扩展其版本依赖.

RQ2. 如何在考虑版本依的情况下得到最佳演化方案?为了满足不断变化的用户需求,考虑到每个可行的演化计划都有特定的成本,例如货币成本、服务停机时间等,演化方案需要满足一定的约束条件.同时,在版本依赖的约束下,应在有限的时间内得到最佳的演化方案.针对 DevOps 流程中的部署需求,需要在考虑版本依赖的情况下,正确执行相关指令.

RQ3. 如何在版本依赖的约束下自动执行演化方案?由于多版本共存微服务系统的复杂性,该系统应具有自适应性[6]:监视系统的运行时状态、决定何时以及如何演化、自动执行演化方案.在演化过程中,应保证多版本共存系统中请求路由的正确性,并满足系统中各个微服务实例的版本依赖.此外,为了降低系统维护的复杂度,需要能够自动化处理用户需求变化以及DevOps流程中部署需求,减少人工操作.

本文主要有3点贡献.

1) 提出了版本依赖模型(version dependency model,简称VDM)以解决RQ1的版本依赖建模,该模型很好地解决了迭代部署问题,同时实现了编程框架 Microservice Framework for Microservice System (MF4MS),通过将版本依赖描述集成到源代码中,可以让系统在部署之前实现服务依赖的自动分析,并能够让多版本同时部署在系统中.

2) 提出了一系列自适应算法以解决 RQ2.这些算法在不同的应用场景中,面对不同种类的用户需求变化,在版本依赖的约束条件下,找到近似最优解的演化方案,从而随着用户需求的变化而维持或者提高 QoS;在处理 DevOps 流程中的部署需求时,能够自动计算依赖关系,并得到演化方案以自动执行DevOps流程中部署需求.

3) 开发了基于 MAPE-K 模型[7]的微服务系统架构 Microservice Infrastructure for Microservice System (MI4MS)以解决 RQ3.它采用自控制循环(self-control loop),可以在满足版本依赖的情况下,对不同种类的用户需求变化进行服务系统的运行时监控、演化方案的生成以及演化方案的自动执行.针对DevOps流程中部署需求,能够接收来自开发者的操作指令,在考虑依赖的情况下自动执行相关指令.

通过对现实世界中常见的应用场景的刻画,构建了真实的云边缘环境来进行实验.结果表明:该方法在有着复杂版本依赖的情况下,针对用户需求变化能够进行自适应演化,并且保持 QoS 稳定;针对 DevOps 流程中的部署需求,该方法能够正确处理依赖关系并自动执行相关的DevOps操作,减轻了系统维护的负担.

本文第1节介绍版本依赖模型VDM以及自适应系统的设计.第2节介绍不同场景下的相关演化算法.第3节介绍编程框架和微服务系统架构的具体实现.第4节介绍实验设计以及实验结果.第5节介绍相关工作.最后在第6节进行总结并探讨未来工作.

1 版本依赖模型与自适应系统设计

1.1 版本依赖模型

考虑到服务依赖关系图只能描述特定版本的微服务的 API 之间的调用依赖关系,不适用于不断变化的服务依赖关系进行迭代开发,因此我们提出了基于服务依赖关系图的版本依赖关系模型.

定义 1. $s=\langle I,c,m,v\rangle$服务系统中的服务定义,$s\in S$,S 是系统中的服务集合,其中,

- $I=\{i_1,i_2,\ldots,i_n\}$,服务 s 提供的功能接口集合.每一个接口都表示为 $i_i=\langle f_i,l_i,d_i^{in},d_i^{out}\rangle$,其中,$f_i$ 表示服务的

功能接口 i_i 所提供的非结构化文本形式的功能语义描述,l_i 表示接口所提供的质量约束, d_i^{in} 和 d_i^{out} 分别表示接口输入参数和输出参数的大小(单位 KB).

- c 表示服务器运行服务 s 所需的计算资源(如 CPU、RAM、存储空间、网络带宽总量等).
- m 表示服务 s 在资源 c 的情况下,能够在保证服务质量前提下的最大用户数量.
- v=⟨*MAJOR*,*MINOR*,*PATCH*⟩表示服务的版本号.采用了广泛使用的 Semantic Version 定义:MAJOR 变化表示发生了不能向前兼容的 API 变化,MINOR 变化表示以向前兼容的方式修改、增加了功能,PATCH 表示修复了漏洞且保持向前兼容(https://semver.org/).

现有的大多数微服务框架如 SpringCloud 中的传统调用依赖关系可以描述为⟨s,i⟩,通过对服务名称和接口构建相应的路由表以达到服务之间的相互调用,不具备对指定版本的定向请求.且一旦将某些微服务升级到具有不兼容更改的新版本,其他服务就需要在代码级别上对其进行修改,这增加了开发人员的负担.此外,开发人员必须注意系统中越来越复杂的版本依赖,以确保系统中请求的正确路由.为了达到在不进行代码修改且不重启服务实例的情况下,在运行时对依赖关系进行修改,对传统的依赖关系类型进行了拓展:

定义 2. D={p_v,p_i,p_f}为服务系统中的 3 种依赖类型的集合,表示一个服务对其他微服务接口或者功能的具体调用关系,其中,

- $p_v(s,i,v)$=⟨s,i,v⟩,表示一个服务对服务版本在集合 V 中的服务 s 的功能接口 i 的依赖关系,V={v_1,v_2,…,v_n}.即调用版本集合 V 中任意版本的服务 s 的接口 I.
- $p_i(s,i,L)$=⟨s,i,L⟩,表示一个服务对服务 s 的功能接口 i,且其质量约束在集合 L 中的依赖关系,L={l_1,l_2,…,l_n}.即调用服务 s 任意一个质量约束在 L 中的接口 I.
- $p_f(f,L)$=⟨f,L⟩,表示一个服务对任意具备功能 f 且质量约束在集合 L 中的接口的依赖关系.即表示调用任意一个功能为 f 的接口,且该接口的服务质量约束在 L 中.

在诸如 OpenFeign(https://spring.io/projects/spring-cloud-openfeign)等现有的微服务框架中,微服务之间的 API 调用通常是硬编码在代码中,当出现由于错误修复而导致被调用的微服务发生不兼容升级时,如果不进行代码修改,则无法变更微服务之间的依赖关系,进而导致现有方式无法适应快速迭代开发.而通过 3 种微服务依赖类型,可以灵活地对微服务之间的依赖关系进行刻画,并在处理请求定向时,由于可以不与具体版本的服务进行绑定,有着更广泛的定向选择,能够更好地调整实例的部署情况.此外,针对调用关系 p_v,允许开发者通过版本号表示服务的向前兼容性,从而让系统能够按照版本号进行依赖判断,从而为用户需求选择更加合适的服务.

定义 3. *VDM*={⟨s,v,p⟩|s∈S},微服务系统中的版本依赖模型 VDM 刻画了微服务系统中所有微服务每个版本的依赖关系.P={p_1,p_2,…,p_n},p_i∈D.表示每个服务可能有着多个依赖.⟨s,v,p⟩表示 v 版本的服务 s 对其他服务的依赖关系为集合 P.

VDM 通过 p_v 在传统的服务之间的依赖描述上增加了版本集合,对 SDG 进行了拓展,允许系统将指定版本范围的请求重定向到兼容版本.此外,VDM 还提供了另外两个新的依赖类型 p_i 和 p_f.当尝试请求具有特定 SLA 的服务接口时使用 p_i,系统可以自动将具备 p_i 依赖的请求路由到具有期望 SLA 的实例上,无需指定服务的具体版本,即使微服务经常升级,也能够主动定向到符合的实例上.而 p_f 允许开发人员按功能和预期 SLA 进行功能请求,这进一步分离微服务之间的依赖关系,服务系统可以将请求重定向到具有最佳 SLA 和性能的实例.开发人员无需在微服务的独立迭代开发过程中修改源代码,降低了开发人员的负担.

1.2 自适应系统设计

多版本共存的微服务自适应系统由两部分构成:支持多版本的微服务编程框架 MF4MS 以及微服务系统架构 MI4MS.其中,MF4MS 负责在代码层级对多版本共存提供支持,提供对 3 种依赖关系的配置以及对具备指定依赖关系请求的支持;MI4MS 负责在服务实例层面对服务部署进行针对用户需求变化的自动调控.

1.2.1 编程框架设计

编程框架 MF4MS 需要具备以下功能.

(1) 提供对 3 种版本依赖关系的配置功能,开发者可以通过编写指定格式的配置文件表示该服务的版本

依赖关系.

(2) 提供对服务的功能接口的功能以及服务质量的描述功能,开发者可以通过注解等方式为服务的每个接口注明功能及质量信息.

(3) 提供发起具备任意类型的版本依赖关系的请求功能,考虑到 VDM 对现有的微服务依赖关系进行了拓展,以及已有的编程框架无法满足 VDM 的需求,MF4MS 需要提供相关接口允许在请求过程中携带额外的版本依赖信息.

(4) 提供集成与未集成 MF4MS 服务实例的区分能力.对于未使用该框架的服务实例,系统不应该将其纳入管理范围内,不能够对其进行任何操作.

(5) 支持运行时的依赖修改.运行时能够通过特定接口对服务实例的依赖关系进行更新而不需要修改源代码、重新编译打包、停止旧实例并部署新实例.

此外,编程框架还需要尽量降低开发者对现有代码迁移的负担.

1.2.2 系统架构

MI4MS 旨在借助 MAPE-K 模型让微服务系统具备自适应能力,系统可以自动检测用户需求变化,生成演进计划,并自动执行计划,如图 2 所示.考虑到不同微服务有着不同的运行环境,MI4MS 使用容器运行每个微服务实例,并使用 Kubernetes 搭建边缘侧容器集群.

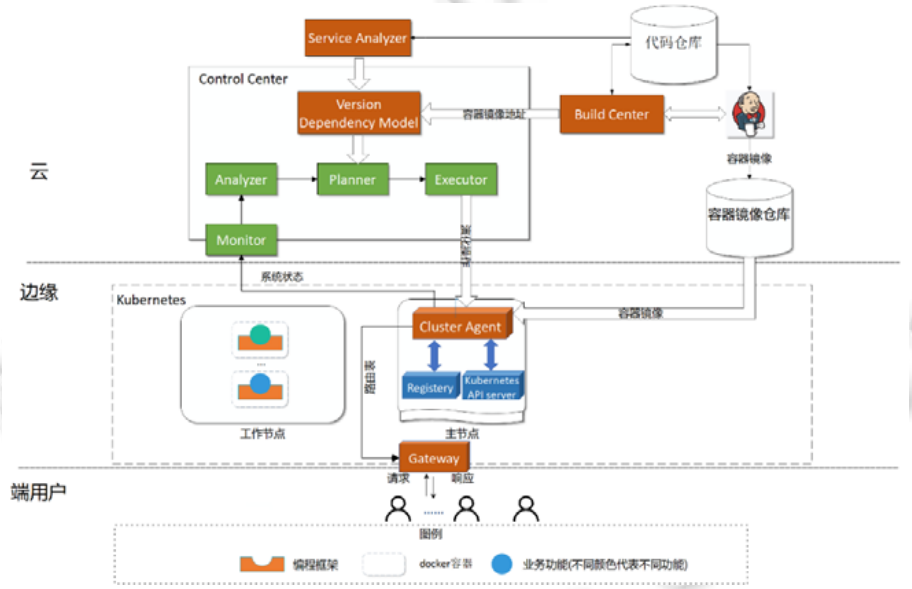


Fig.2 Overview of the MI4MS

图 2 MI4MS 示意图

MI4MS 有 5 个主要组件.

- Control Center.在 MI4MS 中起主要作用,负责控制整个服务系统.它采用了 MAPE-K 模型,并实现了自适应控制循环:在运行时监视系统状态,分析系统当前 QoS 情况以及用户需求变化,通过自适应算法计算出合适的演化方案,最后自动执行方案.该组件应该部署在云端的任意节点上.
- Service Analyzer.旨在分析符合编程框架 MF4MS 规范的微服务源代码,自动从中抽取 3 种类型的服务依赖信息,同时与 Control Center 协作进行版本依赖模型的构建.该组件应该部署在云端节点上.
- Cluster Agent.负责借助 Kubernetes API Server(https://kubernetes.io/docs/concepts/overview/components/#kube-apiserver)和服务注册中心 Registry(https://microservices.io/patterns/service-registry.html)来获取边缘集群的部署状态,并将收集到的信息传递到 Control Center.同时,它需要接收来自 Control Center 的

演化操作指令,并自动执行.该组件应该部署在边缘集群的主节点上.

- Gateway.负责处理来自服务实例和用户的所有请求.通过与Cluster Agent的协作,为具有不同类型依赖关系的请求分配满足依赖的实例并进行请求转发.同时对请求进行记录,为在Control Center提供历史数据从而进行QoS以及用户需求分析.该组件应该部署在任意边缘节点上.
- Build Center.负责对微服务源代码进行自动编译打包,并使用Maven和Jenkins自动进行容器镜像的自动构建以及上传.该组件应部署在任意云端节点上.

通过按固定时间窗口执行自适应控制循环,即可在时间窗口开始时感知到可能的服务质量下降以及用户需求变化,并自动演化以维持或者提高服务质量.MI4MS具备以下主要功能.

(1) 通过Control Center、Service Analyzer以及MI4MS的协作,可以自动从使用了MI4MS的微服务源码中自动抽取出服务接口以及版本依赖信息,从而进行VDM的自动构建,并利用Build Center对新版本的服务进行快速构建交付.

(2) 通过Control Center、VDM以及Cluster Agent能够在满足系统中所有版本依赖时计算出演化方案,并能够自动执行,降低了开发者维护多版本共存的微服务环境的复杂性.

(3) 通过VDM、Cluster Agent以及Gateway的协作,能够在满足服务质量的情况下,对系统中带有版本依赖的请求进行路由,并允许运行时对指定服务的版本依赖进行调整,确保在具有复杂依赖系统中的请求路由正确性.

2 面向用户需求变化及开发者DevOps流程中部署需求的自适应算法

2.1 面向用户需求变化的自适应算法

为了在用户需求发生变化时保持QoS稳定,该算法需要考虑版本依赖和其他约束条件生成最佳演化方案.

2.1.1 抽象描述

定义4. $d=\langle u,p,loc,t\rangle$,用户$u$在时间$t$、地点$loc$,按照依赖关系$p\in D$所提出的需求.服务之间的3种类型的依赖关系$D$在这里也用于表示用户在服务系统中的具体需求,即指定服务、接口、版本的p_v,指定服务、接口、服务质量的p_i以及指定功能和服务质量的p_f.D表示所有的用户需求集合,$d\in D$.

定义5. $e=\langle type,c,loc\rangle$,系统中的服务器节点,其中,$type=\{ES,CS\}$表示服务器节点类型,$ES$代表边缘服务器,$CS$代表云服务器;$c$与定义1中一致,表示该服务器能够提供给服务实例的运算资源;loc表示服务器节点的经纬度坐标.任意两个节点e_i与e_j之间的连接通过延迟(单位毫秒)和带宽(单位Mb/s)表示:$link_{ij}=\langle deloy_{ij},bandwidth_{ij}\rangle$.$E$表示系统中所有服务器节点的集合,$e\in E$.其中,边缘服务器节点和云服务器相比有着更低的延迟和更高的带宽,将服务部署在边缘上能够提升服务质量.但是边缘节点的计算资源有限,需要将一部分实例部署在计算资源相对无限的云端节点上.

定义6. $\tau(s)=\langle s,e\rangle$,系统中服务器节点$e$上服务$s$的一个实例.$Inst$为系统中所有服务实例的集合,$\tau(s)\in Inst$.

定义7. $r(d)=\langle \tau(s),i\rangle$,针对用户需求$d$的请求定向状态,表示将用户需求$d$转发到服务$s$的实例$\tau(s)$的接口$i$上.其中,$DS$表示系统中所有用户需求的请求定向状态,$r(d)\in DS$.

定义8. $\Theta(t)=\langle S(t),E(t),D(t),Inst(t),DS(t)\rangle$,微服务系统在时刻$t$的系统状态,由在时刻$t$的服务集合$S(t)$、服务器节点集合$E(t)$、用户需求集合$D(t)$、服务实例集合$Inst(t)$以及需求请求定向状态集合$DS(t)$构成.

定义9. $OP=\{Switch,Add,Remove\}$,系统演化类型的基本操作集合,其中,

- $Switch(d,\tau_i(s_m),i_j,\tau_j(s_n),i_k)$,将用户需求$d$的请求定向状态$r(d)$从服务$s_m$的实例$\tau_i(s_m)$的接口$i_j$转移到服务$s_n$的实例$\tau_j(s_n)$的接口$i_k$上,其中,$m$与$n$,$j$与$k$可能相同.
- $Add(\tau(s),e)$,在服务器节点e上部署一个新的服务s的实例$\tau(s)$.
- $Remove(\tau(s))$,删除服务s的实例$\tau(s)$.

问题定义. 一个服务系统以时间θ为演化时间间隔,通过一系列演化操作$O=\{op|op\in OP\}$,从时刻t的状态演化到时刻$t+\theta$的状态可以表示为

$$\Theta(t)\xrightarrow{O}\Theta(t+\theta) \tag{1}$$

为了最小化用户请求的平均响应时间和系统演化的成本,可将优化目标表示为如下公式(其中,$rt(\cdot)$表示响应时间,其定义为延迟和请求输入/输出数据传输时间的总和).

$$\begin{aligned}&\min\left\{\begin{array}{l}\dfrac{\sum_{d}^{D}rt(d)}{|D|}\\ \sum_{op}^{O}cost(op)\end{array}\right\}\\ &\text{s.t.}\left\{\begin{array}{ll}Q(\tau(s))\geqslant Q(u_i,d_j), & \forall d_j\in D(t+\theta)\\ \sum_{\tau}^{Inst^{e_k}}r(\tau)\leqslant r_{\max}(e_k), & \forall e_k\in E(t+\theta)\\ 1\leqslant ns(\tau)\leqslant ns_{\max}(\tau), & \forall\tau\in Inst(t+\theta)\end{array}\right.\end{aligned} \tag{2}$$

第1个约束条件确保选定的服务实例可以满足每个用户需求预期从服务中获得的质量.第2个约束条件确保一个服务器节点上所有服务实例消耗的总计算资源不超过该服务器节点所提供的最大计算资源.注意:考虑到云服务器节点相对于边缘节点有着无限计算资源,当边缘节点的计算资源无法再部署新的服务来满足普通用户需求时,系统将在云节点上部署实例以确保用户需求得到满足.最后一个约束条件确保一个服务实例在当前时刻同时服务的用户数量不能超过该实例可以同时服务的最大用户数量.此外,为了确保服务能够正常运行,还应该考虑服务的版本依赖.

2.1.2 算 法

由于无法对未来可能出现的用户进行精确预测,系统无法在规划阶段计算出下一时刻的请求定向状态$DS(t+\theta)$,因此该算法分为两个阶段:规划阶段和运行阶段.规划阶段算法负责生成服务实例重部署演化方案,同时,针对每一类型的版本依赖请求$p\in D$生成路由规则;运行阶段负责利用规划阶段生成的路由规则对系统中所有带版本依赖的请求计算出具体的请求定向状态$r(d)$.

规划阶段,算法接收上一时刻的系统部署状态$\Theta(t)$,输出由*Add*和*Remove*组成的演化方案以及一组路由规则.路由规则描述了对于系统中的每一个具体的依赖关系p应该由系统中服务s的一个具体实例满足,而不会具体到是哪个用户或者实例发出的请求以及哪个实例进行响应.该算法采用如算法1所示的贪婪算法.

算法1. 规划时重部署演化方案及路由规则生成算法.

Input:*os*:上一时刻的系统部署状态.

Output:演化方案以及路由规则.

1: $S\leftarrow getServices(os)$, $N\leftarrow getNodes(os)$, $D\leftarrow getDemands(os)$
2: $unDeployedSvc=\varnothing$, $rules=\varnothing$
3: $ns\leftarrow createEmptyDelopStatus(\cdot)$
4: **for** n in N **do** //以服务器节点为单位进行规划
5: $D_n\leftarrow getNodeDemands(D,n)$, $S_n=\varnothing$
6: **while** $size(D_n)\neq 0$ **do** //优选选择以最少资源服务最多用户的服务
7: $s\leftarrow pickOneService(D_n,S)$
8: $D_s\leftarrow getMetDemands(D_n,s)$
9: $S_n=S_n\cup\{s\}$, $D_n=D_n\backslash D_s$
10: **end while**
11: $T_s\leftarrow buildMiniSvcTree(S_n,S)$ //构建满足服务间版本依赖的最小服务集合
12: $rules\leftarrow rules\cup getRules(T_s)$
13: $S_n\leftarrow getAllServices(T_s)$

14: *instSizeMap*←*calcInstNum*(S_n,D_n) //计算满足给定用户需求所需的实例数量
15: **while** (*s*←*getNextSvc*(T_s))≠*null* **do** //优先部署不被其他服务调用的服务
16: **if** *deployInsts*(*ns*,*n*,*s*,*instSizeMap*[*s*])=true **then**
17: S_n←S_n\{*s*}
18: **end if**
19: T_s←T_s\{*s*}
20: **end while**
21: *unDeployedSvc*←*unDeployedSvc*∪S_n
22: **end for**
23: **for** *s* in *unDeployedSvc* **do**
24: **if** *otherNodesCanSupply*(*s*)=false **then**
25: *deployOnMostCloseNode*(*s*,*ns*) //部署在最近的,有着足够计算资源的节点上
26: **end if**
27: **end for**
28: **return** *calcDiff*(*ns*,*os*), *rules*

该算法的基本思想是,尝试以最低的演化代价为每个边缘节点都提供最低的平均响应时间.因此,算法 1 先对每个节点进行规划(第 4 行~第 22 行),然后再针对无法满足的用户需求进行全局规划(第 23 行~第 27 行).第 6 行~第 10 行针对所有的用户需求,在不考虑服务之间的版本依赖的情况下,挑选出能够以最少计算资源服务最多用户的服务.*pickOneService*(·)遍历服务集合,并返回所需计算资源以及能够满足的用户数量的比值最小的服务.*getMetDemand*(·)在挑选出服务 *s* 能够满足的用户需求集合时,会自动扩展版本依赖关系,通过版本号中的 Major 来判断不同版本服务接口之间的兼容性,并允许使用兼容版本满足用户需求.*buildMiniSvcTree*(·)用于从服务集合 *S* 中计算出满足服务集合 S_n 内服务依赖的最小服务集合.通过对 S_n 中的所有服务按照广度优先依次选择满足最多用户且消耗较少资源的服务,已经出现在集合 S_n 中的服务优先选中.路由规则为树的边缘与叶子节点组成的键值对集合.

第 14 行的 *calcInstNum*(·)通过每个版本的服务的最大用户数量与分配给该服务的用户数量,计算出集合中的每个服务需要部署的实例数量.考虑到调用一个服务接口时,该接口可能会多次调用其他接口,该函数利用上一时间窗内的 Gateway 存储的请求转发历史数据,计算出每个服务接口被调用一次导致的调用其他服务接口的平均次数最为调用,并在计算服务之间的调用时,乘上该系数.例如:用户 *A* 请求了服务 s_j 的接口 i_m,调用一次该接口会调用服务 s_k 的接口 i_n 平均 2.8 次,则服务 s_k 满足的用户数量增加 2.8.在存在着更长的调用链时,以此类推.在第 15 行~第 20 行部署实例时,*getNextSvc*(·)返回服务集合中没有别其他服务调用且能够利用最少资源服务最多用的服务.如果该节点上没有足够的资源部署剩余服务,则部署到其他节点(第 23 行~第 27 行).

运行阶段,算法通过查询规划阶段生成的路由规则,为每个带版本依赖的请求选择合适的服务,并在当前系统中选择一个距离用户最近,且同时服务的用户数量不超过最大数量的实例作为目标实例.

2.2 支持DevOps流程中部署需求的自适应算法

系统中复杂的服务依赖,让服务的部署、删除、版本升级等 DevOps 流程中相关的操作变得更加复杂,开发人员需要耗费大量的精力来确保 DevOps 过程中系统的正确性.该部分自适应算法考虑了 DevOps 过程中的常见操作,能够在确保服务依赖被满足的情况下,自动执行相关指令.本文在 DevOps 自适应过程中,仅仅考虑 DevOps 流程中部署需求,不涉及到用户需求.

2.2.1 抽象描述

考虑到不同场景下的复杂需求,每个不同的操作均提供了响应的布尔值类型的标记位,用于表征是否考虑服务之间的依赖关系.由于部分操作对原有系统中稳定的依赖关系具有一定的破坏性,如删除一个被其他服务依赖的服务实例,这里仅遵守 DevOps 流程中部署的操作指令,不考虑因该操作指令导致的依赖缺失.

定义 10. O^I={*Delete*(*τ*(*s*),*flag*),*Deploy*(*s*,*e*,*flag*)},针对服务实例的 DevOps 操作集合,其中,

- *Delete*(*τ*(*s*),*flag*),删除服务 *s* 的指定实例 *τ*(*s*).*flag* 为标记位,当 *flag* 为真时,表示若该实例依赖的服务没有被其他服务使用,则一起删除;当 *flag* 为假时,表示仅删除指定实例.
- *Deploy*(*s*,*e*,*flag*),在服务器节点 *e* 上部署一个服务 *s* 的实例.*flag* 为标记位,当 *flag* 为真时,若系统中的服务实例无法满足该服务的版本依赖,则部署上相关服务以确保依赖得到满足;当 *flag* 为假时,表示仅部署指定实例.

定义 11. O^S={*Upgrade*(*τ*(*s*),*v*,*flag*),*Upgrade*(*s*,*v*,*flag*)},针对服务的 DevOps 操作集合,其中,

- *Upgrade*(*τ*(*s*),*v*,*flag*),将服务 *s* 的指定实例 *τ*(*s*)升级到版本 *v*.*flag* 为标记位,当 *flag* 为真时,同时考虑相关依赖,删除原版本所使用的相关实例并部署新版本缺失的服务;当 *flag* 为假时,仅进行指定实例的版本替换.
- *Upgrade*(*s*,*v*,*flag*),将系统中所有的服务实例 *s* 升级到版本 *v*.

定义 12. O^D={*Change*(*s*,*P*,*flag*)},针对服务依赖的 DevOps 操作集合,其中,$P=\{p_1,p_2,\ldots,p_n\}$为新的依赖描述,表示将系统中的服务 *s* 的依赖关系更新为 *P*.*flag* 为标记位,当 *flag* 为真时,自动对系统中已存在的服务实例进行依赖更新以及相关依赖服务的自动部署;当 *flag* 为假时,仅替换依赖.

2.2.2 支持服务实例相关的 DevOps 操作的自适应算法

针对服务实例的删除操作 *Delete*(*τ*(*s*),*flag*),当 *flag* 为真时,算法 2 通过对已有服务实例之间的依赖分析,筛选出仅被服务实例 *τ*(*s*)调用的其他实例,并一起删除.该算法的主要时间复杂度集中在判断每个服务实例所依赖的其他实例有哪些.设系统中实例数量为 *m*,则该算法的时间复杂度为 O(*m*(*m*−1)).

算法 2. 实例删除算法 *delete*(*τ*,*flag*).

Input:实例 *τ*,标记位 *flag*.

Output:演化操作集合 *opList*.

```
1:  opList←{deleteJob(τ)}
2:  if flag=true then
3:      Inst←getDependInsts(τ)
4:      for i in Inst do
5:      if usedByOtherInst(i)≠true then
6:      opList←opList∪delete(i,flag)   //迭代删除没有被使用的依赖
7:  end if
8:  end for
9:  end if
10: return opList
```

针对服务实例的部署操作 *Deploy*(*s*,*e*,*flag*),当 *flag* 为真时,算法 3 通过对已有服务实例以及目标服务 *s* 的版本依赖分析,对于能够被直接满足的依赖,则直接使用已有实例;否则,从服务集合 *S* 中选择能够满足依赖的服务进行部署.设系统中实例数量为 *m*,则时间复杂度为 O(*m*).

算法 3. 实例部署 *deploy*(*s*,*e*,*flag*)算法.

Input:服务 *s*,服务器节点 *e*,标记位 *flag*.

Output:演化操作集合 *opList*.

```
1:  opList←{deployJob(s,e)}
2:  if flag=true then
3:      D←getDependencies(s)
4:  for d in D do
5:  if depSatisfied(d)≠true then
```

6: s_d←*pickOneService*(*s*,*d*)
7: *opList*←*opList*∪*deploy*(s_d,*e*,*flag*)
8: **end if**
9: **end for**
10: **end if**
11: **return** *opList*

2.2.3 支持服务相关的 DevOps 操作的自适应算法

针对服务相关的实例升级操作 *Upgrade*(*τ*(*s*),*v*,*flag*)的算法 4,若 *flag* 为假,则仅删除原有实例并部署新版本;若 *flag* 为真,则确保相关依赖能够被正确删除、部署.设系统中实例数量为 *m*,则时间复杂度为 O(m^2).

算法 4. 实例升级 *upgrade_inst*(*τ*(*s*),*v*,*flag*)算法.

Input:服务 *s*,实例 *τ*,目标版本 *v*,标记位 *flag*.

Output:演化操作集合 *opList*.

1: *opList*←∅
2: *e*←*getInstNode*(*τ*)
3: s_n←*getSvcByVer*(*s*,*v*)
4: *opList*←*opList*∪*deploy*(s_n,*e*,*flag*) //部署新版本服务实例
5: *opList*←*opList*∪*delete*(*τ*,*flag*) //移除旧版本服务实例
6: **return** *opList*

针对服务升级操作 *Upgrade*(*s*,*v*,*flag*),将按照算法 4 先以实例为基础对系统中服务 *s* 的全部实例进行升级操作,再将目标版本的服务标记为不可用,防止在下次自演化时重新进入系统.

2.2.4 支持服务依赖相关的 DevOps 操作的自适应算法

针对服务依赖变更操作 *Change*(*s*,*P*,*flag*),算法 5 在 *flag* 为真时,自动判断系统中服务 *s* 的所有实例的依赖满足情况,并按需进行实例的部署、删除.设系统中实例数量为 *m*,则时间复杂度为 O(m^2).

算法 5. *change*(*s*,*P*,*flag*)算法.

Input:服务 *s*,新依赖关系集合 *P*,标记位 *flag*.

Output:演化操作集合 *opList*.

1: *opList*←∅
2: *Inst*←*getSvcInsts*(*s*)
3: **for** *inst* in *Inst* **do**
4: **for** *p* in *P* **do**
5: **if** *depSatisfied*(*p*)≠true **then**
6: *e*←*getInstNode*(*τ*)
7: s_d←*pickOneService*(*s*,*p*)
8: *opList*←*opList*∪*deploy*(s_d,*e*,*flag*)
9: **end if**
10: **end for**
11: **end for**
12: *opList*←*opList*∪*deleteUnusedInst*(·) //删除没有被使用的实例
13: **return** *opList*

3 多版本共存的微服务自适应系统实现

3.1 编程框架MF4MS实现

为了尽可能地降低开发者地负担,需要尽可能少地对原有代码进行变更.同时,为了方便系统对所包含的版本依赖关系进行自动分析,MF4MS 提供了配置文件、注解以及函数接口相结合的方式.

(1) 提供指定格式的配置文件,支持 3 种类型的版本依赖以及多依赖的描述,如图 3(a)所示.

(2) 提供@MFuncDescription 注解,用于对功能接口功能以及质量信息的描述,如图 3(b)中(1)所示.

(3) 提供 MVerRequestUtils.request 函数,用于携带版本依赖描述的请求,如图 3(b)中(2)所示.

(4) 提供@MClient 注解,用于在运行时区分出使用了该框架的服务实例,并提供指定的依赖变更接口,运行在运行时接收相应指令并进行依赖变更.

```yaml
mvf4ms:
  version: 1.0.1
  dependencies:
    - name: dependency1
      dependence:
        - id: navigation
          serviceName: SampleGaoDe
          patternUrl: /navigation
          versions:
            - 1.1.1
            - 1.2.2
        - id: weather
          function: weather
          slas:
            - 2
        - id: pay
          function: pay
          slas:
            - 2
            - 3
```

(a) MF4MS 配置

```java
@Controller
public class MainController {

    private Logger logger = LogManager.getLogger(this);

    @PostMapping(path = "/taxi")
    @ResponseBody
    @MFuncDescription(value = "taxi", level = 1)   (1)
    public MResponse weather(
        @RequestBody MResponse params, HttpServletRequest request) {

        MResponse p = new MResponse();
        MResponse response =
            MVerRequestUtils.request("weather", p, RequestMethod.POST, request);
        // ...                                                    (2)

        return response;
    }
}
```

(b) 集成了 MF4MS 的 Controller

Fig.3 An example of the MF4MS integration

图 3 MF4MS 使用实例

3.2 系统架构MI4MS实现

3.2.1 依赖模型自动构建

开发人员在开发微服务时,需要在代码级别与 MF4MS 集成,每个微服务的每个版本的版本依赖应配置到配置 application.yaml 中,如图 3(a)所示.同时,在每个接口函数上,需要通过@MFuncDescription 注解添加该 API 的功能和 SLA 描述,如图 3(b)(1)所示.和服务有关的其他信息,如每个版本的资源使用情况,服务的最大用户数量和源代码仓库等,应包含在服务进入系统时的服务描述文件中(第 4.2.2 节).

Service Analyzer 组件提供了 getMicroserviceInfo 的 Restful 接口,负责接收从 Control Center 传递的指定代码仓库中解析该服务的依赖以及接口等信息.首先,利用 JGit 将指定的微服务源代码克隆到本地;然后,针对每个版本标签分别进行代码解析.根据 Spring Cloud 框架开发的微服务的特性,通过遍历文件来查找 application.yaml 文件,并使用 Yaml 工具对该文件进行解析文,从而获取其中包含的版本依赖信息(mvf4ms.dependencies).此外,它利用 JavaParser 对代码中所有的控制类(controller)进行语法解析,获取其中包含的@MFuncDescription 以及对应的接口名称、路径等信息.最后,将这些信息与配置文件中的微服务的名称、开放端口以及版本号信息一起返回.Control Center 接受到分析结构后,将其中所包含的版本依赖信息记录,并合并到已有的版本依赖模型中.

3.2.2 服务注册流程

服务开发者主动通过 Control Center 的 register 接口将服务注册到系统中,并提供包括服务名称(serviceName)、源代码仓库地址(gitUrl)、每个版本的服务资源消耗情况(resMap)以及每个版本能同时服务的最大用户数量,具体格式如图 4 所示.Control Center 接受到服务注册请求后,会将该服务的代码仓库地址发送到

Service Analyzer 并构建依赖模型.然后查询该服务的每个版本在容器镜像仓库中是否已经存在镜像文件:如果不存在,则发送镜像构建任务到 Build Center.

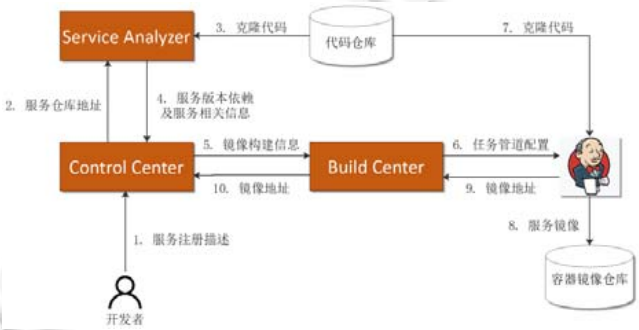
Build Center 通过 JenkinsServer 提供的 Java API 来控制 Jenkins,并提供了 autoBuild 接口,接收微服务的原源码仓库地址,服务版本号以及微服务名称,然后将参数转译为 XML 形式的管道(pipeline)配置文件,并通过 JenkinsServer 的 createJob 接口将 XML 文件发送到 Jenkins 构建出管道.该管道包含 3 个任务.

- 从指定的源码仓库克隆指定版本标签的源代码;
- 利用 Maven 进行自动编译;
- 利用 docker 构建镜像容器并推送到镜像仓库.

全部任务完成后,Build Center 返回镜像文件地址到 Control Center,并保存入库.具体工作流程如图 4(b).

```
"serviceName": "SampleSvc",
"gitUrl": "http://sample/samplesvc.git",
"resMap": {
    "1.1.1": {
        "cpu": 100,
        "ram": 200,
        "bandwidth": 1000
    },
    "2.1.1": {
        "cpu": 100,
        "ram": 200,
        "bandwidth": 1000
    }
},
"plotMap": {
    "1.1.1": 100,
    "2.1.1": 150
}
```

(a) 服务注册信息



(b) 服务注册流程

Fig.4 Service registry description and process

图 4 服务注册描述信息及流程

3.2.3 多版本共存路由

该部分由 Gateway 和 Cluster Agent 协同工作完成,如图 5 所示.

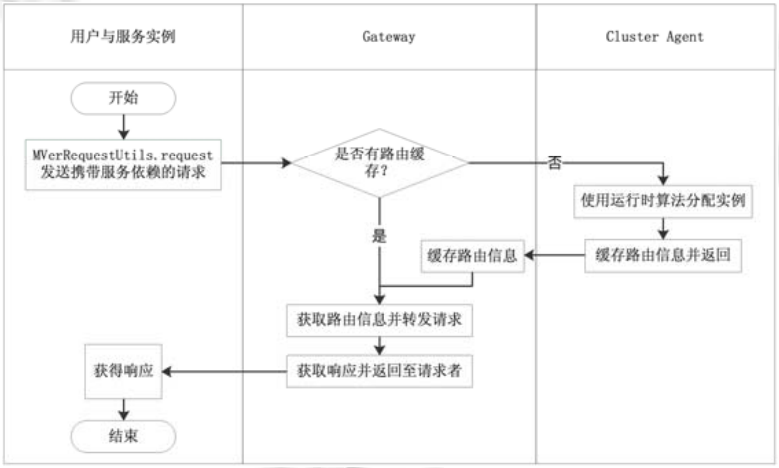


Fig.5 Request routing process in multi-version coexisting microservice system

图 5 多版本共存的微服务系统中的请求路由流程

对于实例和用户的所有请求,开发人员都需要使用 MF4MS 提供的 MVerRequestUtils.request,并携带在配置文件中定义的唯一依赖项 ID 进行请求的发送,如图 2(b)所示.由于不同接口之间的数据格式不是本文的重点,因此提供了一个以键值映射表为基础的 MResponse 类型来保存所有参数或返回值.所有带有版本依赖的请求

均由MVerRequestUtils.request发送到Gateway,然后,Gateway根据请求者以及携带的版本依赖进行请求重定向.如果Gateway中没有针对该请求者的路由缓存,则发送请求到Cluster Agent进行响应实例的分配.Cluster Agent通过运行阶段算法(第3.1.2节)为该请求者的该种版本依赖分配响应实例,分配完成后缓存分配信息,并将URL返回到Gateway.之后,Gateway根据路由信息重定向请求并缓存该路由信息.

4 实 验

本文的实验在由多个云服务器搭建成的云-边缘实验环境中进行,其中,5个8vCPU和16GB RAM的AWS EC2实例通过Kubernetes 1.18.2作为边缘集群,各个节点之间的延迟小于1ms,带宽为1000Mb/s.另外,两台AWS EC2实例作为云服务器节点,边缘集群中的各个节点和云端之间的节点延迟为20ms,带宽为1000Mb/s.Control Center、Service Analyzer和Build Center均部署在云服务器节点上.Cluster Agent部署在Kubernetes边缘群集的主节点上,Gateway部署在每个边缘服务器节点上.

实验中使用了根据现实中的打车、购物和支付场景实现的服务集1和服务集2.集合1有6个服务,每个服务具有0~2个依赖,且依赖关系不超过两层,即服务*A*依赖于服务*B*,服务*B*不依赖其他服务;服务集合2对服务集1进行了拓展,新添了4个新服务,且每个服务至少具有3层依赖.在服务集1和服务集2中,每个服务都有2~3个不同的接口,每个接口的输入和输出数据的大小随机在1KB~20KB之间.每个服务都有2~3个版本,且同一服务的不同版本的版本依赖与其他版本不同.集合1和集合2中均包含3种依赖类型,且集合中的每个服务的同时最大用户数量在100~300之间.

实验中有2 000个模拟用户,均匀分布在5个边缘服务器上.每个用户每5s~10s会发送具备版本依赖的请求到最近边缘服务器节点上的Gateway实例.在本次实验中,平均响应时间和失败用户需求数量作为评估指标,同时,部分实验还采用由服务停机总时长除以总运行时间得到的服务可用性来评估系统性能.在计算服务可用性时,仅对受影响的服务进行计算.

针对用户需求变化,服务升级以及用户新需求的提出是现实中较为常见的应用场景.在服务升级场景中,用户需求会随着服务的升级而不断发生变化;在用户新需求提出的场景中,开发者通过不断更新服务来满足用户的新需求.这两种场景均能够很好地表征用户需求的变化,因此我们分别对这两种场景进行实验.考虑到现有算法不适合该问题,因此,本文通过对现实世界中的两种常见场景的模拟,对该系统进行评估.针对开发者敏捷DevOps流程中的部署需求,我们针对多种不同的需求类型分别进行了实验进行对比.本文中所提及的新需求均指系统服务库中存在服务能够满足的或者开发者将要推出新服务能够满足的用户需求,没有对应服务能够满足的用户新需求不在本文考虑范围.

4.1 普通用户需求变化

4.1.1 场景1:服务升级

考虑到服务升级是现实世界中较为常见的场景,我们分别使用实验集合1和集合2对该种场景进行了模拟实验.系统中的普通用户被分为3类:在服务的新版本发布后立刻更新相关需求至最新版本;在新版本发布后的随机时间内保持原需求不变,然后变更为最新需求;相关需求不随服务更新变化.系统自演化的时间窗口设置为5分钟,具体实验结果如表1及图6(a)、图6(b)所示.

Table 1 Service availability of Experiment 1~Experiment 3

表1 实验1~实验3的服务可用性

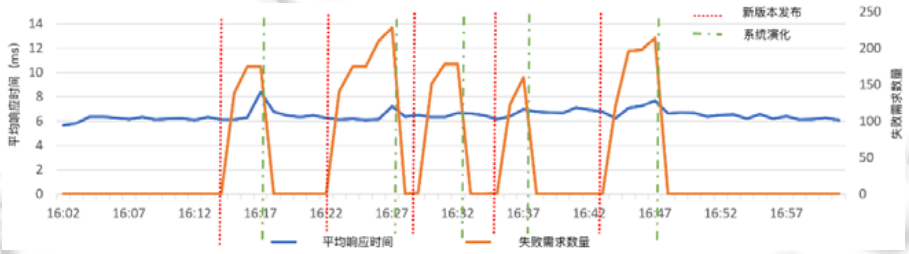
实验	服务失效总时长(分钟)	服务可用性(%)
1	3.0	95
2	2.5	95.83
3	28	53.33

实验结果表明:当系统检测到失败用户需求的出现后,系统会进行自动演化以满足失败的用户需求,进而提高QoS.系统演化结束后,针对之前失败的用户需求,部署了相关的服务实例,并确保它们的版本依赖得到满足,

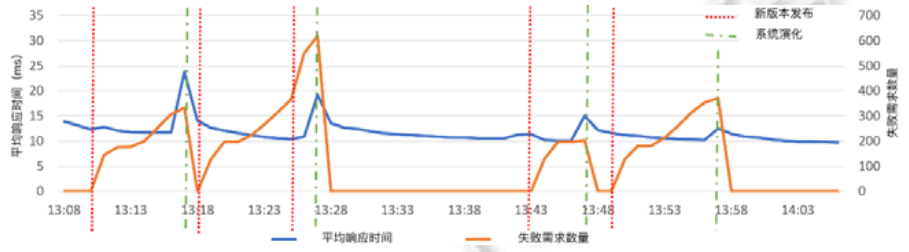
之前无法得到满足的用户需求数量降为 0.使用了具有简单版本依赖的服务集合 1 的实验 1 与复杂版本依赖关系的服务集合的实验 2 在系统演化之后,未满足用户需求数量均降到 0,且用户的平均响应时间保持相对稳定.需要注意的是:在每次演化过程中,平均响应时间均为先增加后减少.因为在每次演化之后,系统需要从 Gateway 向 Cluster Agent 发送请求以为每个需求重新计算路由信息,导致了平均响应时间的增加.在 Gateway 中缓存路由信息后,Gateway 无需再向 Cluster Agent 发送请求,因此平均响应时间降低.此外,表 1 中的服务可用性表明:在不同的服务依赖复杂程度下,该系统均能够很好地针对用户需求变化进行自演化,且具有较高的服务可用性.

为了研究不同时间窗口的影响,设置了具有不同演化时间窗口大小的实验 3.实验 3 与实验 2 的设置包括服务升级时间和用户设置等基本相同,但时间窗口设置为 10 分钟.实验结果显示在图 6(c)和表 1 中.结果表明:随着时间窗口的增大,系统需要更多的时间来发现未满足的需求,因此服务的可用性降低了很多.但较大的时间窗口会降低相同时间内系统演化的次数.实验 2 系统共演化了 5 次,而实验 3 系统演化了 4 次,考虑到每次演化过程都会导致平均响应时间的波动,当时间窗口设置较大时,其他用户受到系统演化的影响就越小.

(a) 实验 1:服务集合 1,演化时间窗口为 5 分钟



(b) 实验 2:服务集合 2,演化时间窗口为 5 分钟



(c) 实验 3:服务集合 2,时间窗口为 10 分钟

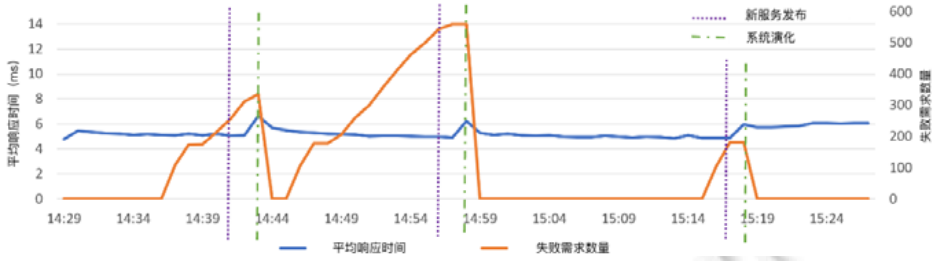
Fig.6 Average response time and count of failed demands in service upgrade scenario

图 6 服务升级实验场景下的平均响应时间和失败需求数量

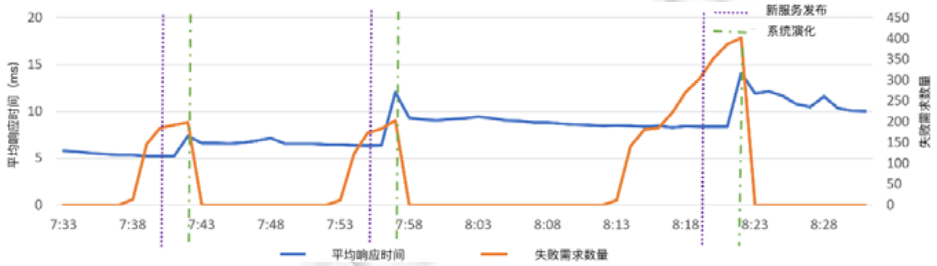
4.1.2 场景 2:新用户需求

该实验对现实世界中的另一种常见情况进行了模拟:用户提出新要求后,开发者发布了新版本服务以满足用户的新需求.该场景下的用户设置与实验 1~实验 3 不同,部分用户会主动提出新需求,且这些需求无法通过系统中的服务集合满足.在新需求出现后,能够满足这些新需求的服务在随机的时间内注册到系统.分别使用服务

集 1 和服务集 2 进行了实验 4 和实验 5,时间窗口为 5 分钟.由于发布新服务的时间会严重影响服务的可用性,因此仅采用平均响应时间和失败的用户需求数量进行评估.结果显示在图 7 中.



(a) 实验 4:服务集合 1,演化时间窗口为 5 分钟



(b) 实验 5:服务集合 2,演化时间窗口为 5 分钟

Fig.7 Average response time and count of failed demands in new demands scenario

图 7 新需求实验场景下的平均响应时间和失败需求数量

结果表明:无论是在具备简单的服务依赖的情况或是复杂的服务依赖的情况下,MF4MS 均能够在新服务注册到系统后,稳定平均响应时间并满足用户的新需求.当开发者将能够满足之前无法满足的用户需求的新服务后,系统会在一个演化时间窗口内快速响应,在考虑服务之间的复杂版本依赖关系的同时,快速对服务进行编译打包以及部署,保持了 QoS 的稳定.

目前已有的一些方法,如 APP-bisect[8]、VMAMVS[9]等,虽然考虑到了微服务之间的依赖关系,且能够在一定程度上对系统中出现的问题进行自动修复,但这些方法不具备针对用户需求变化以及开发者敏捷 DevOps 的自适应能力,无法用于对比实验;而像文献[10–12]等提出的方法尽管具备一定的自适应能力,能够针对系统中的性能问题进行横向拓展,却也无法处理用户的需求变化及 DevOps 流程中的部署需求.此外,这些方法无法处理微服务多版本共存以及微服务之间依赖关系,亦无法作为对比实验.

4.2 DevOps流程中部署需求自适应实验

该部分实验选取了实验集合 1 和集合 2 中的部分服务作为 DevOps 操作的对象,并通过 DevOps 指令数量、实例部署数量、实例删除数量、是否需要人工参与、是否支持带依赖部署以及是否支持多版本共存、是否修改源码且停止服务实例这 7 个方面对第 3.2 节给出的各种操作在考虑依赖关系的情况下进行实验对比.其中,DevOps 指令数量是指由开发者发送给系统或者工具的指令数量;实例部署、删除数量是指完成对应的 DevOps 操作过程中部署、删除了多少个实例;是否需要人工参与是指在执行 DevOps 操作过程中是否需要参与进行决策、判断等人工操作;是否支持带依赖部署是指该种方法是否能够自动处理依赖关系;是否支持多版本共存是指该种方法是否能够允许同一个服务不同版本同时存在;是否修改源码、停止服务实例表示在服务依赖变更过程中,是否需要人工修改服务源代码,并停止原服务实例.这里选取 Kubernetes 以及 JRO(jolie redeployment optimizer)[13]进行对比实验.其中,Kubernetes 是一套流行的容器管理工具,JRO 是一套考虑了服务之间的依赖关

系的部署工具.

实验 6~实验 13 中均在实验集合 1 和集合 2 中选取相同的具有依赖的服务组作为实验对象,其中,在实验 1 中,从实验集合里选取一个具有 4 个服务依赖的服务 *A*;在实验 2 中,选取服务 *A*、具有 7 个依赖的服务 *B* 以及具有 5 个依赖的服务 *C*,*A*,*B*,*C* 的依赖均不相同.实验 6、实验 7 分别对这两组服务进行服务实例的部署操作;实验 8、实验 9 进行对系统中以满足依赖关系的实例删除操作.

实验 10、实验 11 进行对该实例的升级操作.其中,实验 10 将系统中的服务 *A* 升级到新版本,依赖关系变更为 5 个,其中两个依赖关系和升级前兼容,另外 3 个依赖均与之前不同且没有子依赖;实验 11 除了升级服务 *A* 之外,还将服务 *B* 升级,新增 2 个依赖,服务 *C* 升级,减少 2 个依赖.

实验 12、实验 13 对该实例进行依赖变更操作:实验 12 将服务 *A* 的 1 个依赖关系替换为新依赖;实验 13 除此之外,还将服务 *B* 的 2 个依赖关系以及服务 *C* 的 3 个依赖关系均替换为新依赖.其中,所有的新依赖均不包含子依赖.实验结果见表 2.

Table 2 Experimental results of Experiment 6~Experiment 13

表 2 实验 6~实验 13 的实验结果

实验	方法	DevOps 指令数量	实例部署数量	实例删除数量	是否人工参与	是否支持依赖关系	是否能多版本共存	是否修改源码,停止服务实例
实验 6(部署 *A*)	Kubernetes	5	5	0	是	否	是	否
	JRO	1	5	0	是	是	否	否
	MF4MS	1	5	0	否	是	是	否
实验 7(部署 *B*)	Kubernetes	19	19	0	是	否	是	否
	JRO	3	19	0	是	是	否	否
	MF4MS	3	19	0	否	是	是	否
实验 8(删除 *A*)	Kubernetes	5	0	5	是	否	是	是
	JRO	5	0	5	是	是	否	是
	MF4MS	1	0	5	否	是	是	是
实验 9(删除 *B*)	Kubernetes	19	0	19	是	否	是	是
	JRO	19	0	19	是	是	否	是
	MF4MS	3	0	19	否	是	是	是
实验 10(升级 *A*)	Kubernetes	6	4	3	是	否	是	是
	JRO	4	4	3	是	是	否	是
	MF4MS	1	4	3	否	是	是	是
实验 11(升级 *B*)	Kubernetes	12	8	7	是	否	是	是
	JRO	10	8	7	是	是	否	是
	MF4MS	3	8	7	否	是	是	是
实验 12(依赖变更 *A*)	Kubernetes	6	3	3	是	否	是	是
	JRO	4	3	3	是	是	否	是
	MF4MS	1	2	2	否	是	是	否
实验 13(依赖变更 *B*)	Kubernetes	20	10	10	是	否	是	是
	JRO	13	10	10	是	是	否	是
	MF4MS	3	7	7	否	是	是	否

从实验 6、实验 7 可以看出:Kubernetes 由于不支持服务之间的依赖关系,导致在部署带依赖的服务时,需要人工参与来判断服务的依赖是否满足以及需要额外部署哪些服务.第 1 组服务 *A* 有两个依赖,因此一共需要部署 3 个实例;第 2 组服务一共有 16 个不同依赖,一共需要部署 19 个实例.每次部署操作均需要一个 DevOps 指令(Kubernetes 可以通过配置文件的方式将所有操作配置到一个文件中,这里我们依然计算指令数量,而不是配置文件数量).而 JRO 以及本文的方法 MI4MS+MF4MS 则能够根据服务依赖描述自动进行依赖部署,仅需要进行 3 次指令部署 3 个服务 *A*,*B*,*C*.

实验 8、实验 9 中,由于 JRO 仅仅支持带依赖关系的实例部署操作,因此 JRO 与 Kubernetes 相同,均需要多次操作指令才能够将指定服务实例以及对应的依赖实例删除.

实验 10、实验 11 中,由于 Kubernetes 支持容器的升级操作,因此在升级服务实例 *A* 时,需要删除一个失效依赖,部署一个新的依赖实例,并对服务 *A* 的实例执行升级操作,一共 6 个操作指令,部署 4 个新实例(新的依赖实

例以及服务 *A* 的新版本)以及删除 3 个实例(失效依赖实例以及服务 *A* 的旧版本).而 JRO 由于支持带依赖部署,因此只需要删除失效依赖实例以及服务 *A* 的旧版本,并部署新版本的 *A* 及其依赖,共 4 个操作指令.

实验 12、实验 13 中,由于 Kubernetes 与 JRO 不具备运行时的依赖变更,因此需要对源码进行修改,停止原实例并部署新的实例.且由于 Kubernetes 不支持依赖关系,JRO 仅支持部署时的依赖关系,因此两种方法均需要多次指令才能够达到最终效果.而本文的方法支持运行时依赖变更且只需要 1 次以及 3 次操作即可.

此外,由于 Kubernetes 不支持依赖关系,JRO 在运行时需要用户主动提供依赖信息,因此均需要人工参与;而 MI4MS 支持对服务依赖关系的自动分析获取,因此不需要人工参与.从实验结果可以看出:MI4MS 能够主动考虑服务之间的依赖关系,对特定的 DevOps 操作降低开发者的负担.

在实验过程中,本文提出的方法需要若干系统相关数据:用户需求分布情况、实例分布情况、服务器节点资源分配情况.其中,实例分配情况与服务器节点资源分配情况均在实例的创建、删除过程中,同步记录到 ControlCenter 中,因此无需重复进行实时的信息采集;用户需求分布情况主要由各个 Gateway 实例负责记录,只需保留每种需求的用户数量即可,传输的数据量较少,对系统的性能影响亦较少.

5 相关工作

近年来研究了多版本共存、演化计划生成和自适应服务系统中的问题,并提出了一些解决方案.

对于多版本共存,文献[14]探索了微服务系统中的自主版本管理.它考虑了应用程序级别和公司级别的微服务版本控制,允许服务的热替换、在不干扰其他服务正常运行情况下的服务升级.通过对微服务版本的自动管理,利用服务迁移等手段,确保系统能够从错误状态快速恢复.文献[8]提出了基于微服务依赖关系图进行系统自我修复的工具 APP-bisect,该工具利用不同微服务不同版本的依赖关系,当服务系统发生性能故障时,通过分析微服务之间的依赖关系,寻找各个微服务之间最佳的版本共存模式,移除所有的其他版本实例,来消除性能故障.文献[9]提出了一种名为 VMAMVS 的解决方案,用于分析微服务之间的依赖关系,监视系统并可视化这些依赖关系.该方法通过在源代码层面上自动解析基于 Java 的微服务,利用特定的微服务框架的注解等关键特征,分析不同微服务之间的调用关系,构建出微服务依赖模型.同时,利用微服务的接口信息构建系统监控指标,结合服务日志分析实现运行时的系统监控,并最终进行可视化操作.文献[15]对微服务架构进行了拓展.通过在服务网关中的智能路由算法,允许多版本共存的微服务系统中的客户端和微服务请求指定微服务的特定版本,在满足用户期望的服务质量的情况下,降低了服务的运行代价.文献[16]提出了基于适配器的方法以消除了同一服务的不同版本之间的接口差异问题.通过对不同版本的相同接口进行分析,针对不能直接兼容的接口,通过动态构建不同版本之间的适配器,允许高版本的服务实例响应低版本的接口.文献[13]提出了 JRO 工具,通过用户在配置文件中对服务之间的描述关系的描述,自动在部署服务实例时确保依赖能够得到满足.但是,该种方法不支持多版本共存.尽管这些工作中考虑了微服务之间的依赖关系,它们专注于特定版本的请求路由、监测、问题溯源及恢复,而不是根据用户需求的变化进行自适应演变.本文通过编程框架 MF4MS 以及系统架构 MI4MS 的协作,能够自动进行微服务之间的版本依赖分析以及建模,并能够对已有的版本依赖主动进行兼容拓展,大大降低了微服务之间版本依赖维护的复杂性,满足了快速迭代开发的需求.

在演化方案生成方面,文献[17]评估了 3 种雾计算(fog computing)场景下的服务重部署算法,在考虑了服务资源使用、服务传播性(service spread)和延迟的优化目标下,计算服务的重部署方案.文献[18]提出了一种服务放置的优化策略,通过将用户使用的服务尽可能地部署到距离用户较近的边缘节点上,改善网络使用率和服务延迟.然而在当前的工作中,服务之间的依赖关系是固定不变的,与现实中不断变化的服务之间的依赖情况不符,并且没有考虑到用户需求的变化.本文通过基于贪婪的算法,在考虑了服务的版本依赖的情况下,能够针对不同场景下的用户需求变化,快速生成较优的系统演化方案.

对于自适应服务系统,文献[10]开发了 MiCADO,通过在服务系统中添加统一的服务编排层,让系统具备根据网络流量变化对系统中的服务进行自动横向拓展以提高系统性能.文献[11,12]使用了 MAPE-K 模型来根据性能自动优化服务部署.通过对系统的运行时监控,根据性能变化来计算出合适的演化方案,并进行方案的自动

执行.但是在这些工作中,没有考虑到复杂的用户需求变化,且忽略了微服务之间的版本依赖.本文通过监控用户需求变化,利用 MAPE-K 模型,结合版本依赖模型,运行微服务版本依赖的不断变化,并能够针对用户需求变化自动执行演化方案来维持或者提升系统服务质量,大幅度降低了需求频繁变化的复杂系统维护的复杂性.

6 总结和下一步工作

本文面向多版本并存的微服务系统提出了针对用户需求变化的自适应方法和系统实现.针对不断变化的微服务之间的版本依赖,提出了版本依赖模型,并实现了依赖模型的自动构建;针对有着复杂版本依赖的用户需求变化的场景,实现了适用于不同需求变化场景的演化算法,能够得到较优的演化方案;针对开发者的 DevOps 流程中的部署需求,提供了几种常用的操作及对应算法,并能够自动考虑依赖关系来降低开发者的负担.为了让微服务系统具备自适应能力,实现了编程框架 MF4MS 和自适应系统结构 MI4MS,在满足版本依赖的情况下,能够自动针对用户需求变化进行演化,并保持服务质量稳定,以及针对 DevOps 流程中的部署需求,降低 DevOps 的复杂程度.

在目前的 MI4MS 设计方面,服务系统内部微服务之间相互调用的东西流量依然需要 Gateway 进行协调,从而避免依赖不满足、服务实例的服务能力不够等情况.和传统的负载算法在服务调用方实现以及点对点通讯方式相比,好处是不再需要额外的模块来确保在每个实例独立负载决策时,全局范围内的约束条件(如每个服务实例的服务能力)依然能够得到满足.在全局约束下,无法完全在实例侧进行负载均衡,需要额外的措施确保每个实例的负载均衡结果满足全局约束,考虑到本文的目标是多版本依赖下的自适应问题,本文直接在 Gateway 中进行相关控制.此外,也可以通过在实例侧增加额外的路由规则缓存,并通过合适的缓存失效机制确保缓存的有效性,从而实现生命周期中的大部分时间内进行点对点连接.

在用户需求方面,虽然本文考量了用户需求变化中新需求的出现等问题,这些新需求依然局限于现有的服务集合,即系统能够自适应的用户新需求必须是现有服务集合中能够满足的需求,否则无法通过部署相关服务实例来满足新需求,未来需要提供合适的基于外部服务库等方式主动为系统用户提供满足新需求的新服务.

此外,由于系统需要收集到全局的信息后才能够进行决策并进行相关演化,导致演化操作往往不能够第一时间被触发.比如规模较大时,系统中可能只有一个比较集中区域内的用户需求发生了剧烈变化,其他地区没有变化,那么此时系统依然需要等待时间窗口,并收集全局所有信息才能够进行自适应演化,导致了不必要的性能影响,因此需要进一步考虑系统的区域划分问题.同时,当规模扩大后,集中式的算法不易处理大数量级别(比如几千服务器节点、几十万服务等)的数据,通过对整个系统进行合适的区域划分,利用区域之间的协同优化,实现区域之间演化窗口的相互独立以及问题的分解.

后续可能的工作:首先,集中式的微服务系统调度将集中了全部的分析、计算工作,浪费了其他节点的资源,同时还需要将所有的数据都收集到一起,有着大量的数据传输,响应速度变慢,后续工作引入去中心化演化,将整个系统区域化,各个区域之间进行独立且协同演化;其次,当出现新需求时,如果系统中现有的服务无法满足,则不会主动给出解决方案,下一步工作会对功能空洞监测以及功能空洞的自动、半自动补全,以在出现无法满足的新需求时依然能够继续维持系统的服务质量;此外,目前的算法仍然需要进一步的提升,才能够达到大规模系统的自适应需求.

References:

[1] Gaur A, Scotney B, Parr G, McClean S. Smart city architecture and its applications based on IoT. Procedia Computer Science, 2015, 52:1089−1094.

[2] Feng ZY, Xu YW, Xue X, Chen SZ. Review on the development of microservice architecture. Journal of Computer Research and Development, 2020,57(5):1103−1122 (in Chinese with English abstract).

[3] Wu HY, Deng WJ. Research progress on the development of microservices. Journal of Computer Research and Development, 2020, 57(3):525−541 (in Chinese with English abstract).

[4] Ma S, Fan C, Chuang Y, Lee W, Lee S, Hsueh N. Using service dependencygraph to analyze and test microservices. In: Proc. of the IEEE Annual Int'l Computer Software and Applications Conf. (COMPSAC). 2018. 81−86.

[5] Satyanarayanan M. The emergence of edge computing. Computer, 2017,50(1):30−39.

[6] Weyns D. Software Engineering of Self-adaptive Systems: An Organised Tour and Future Challenges. In: Handbook of Software Engineering. Springer-Verlag, 2017.

[7] Kephart JO, Chess DM. The vision of autonomic computing. Computer, 2003,36(1):41−50.

[8] Rajagopalan S, Jamjoom H. App-Bisect: Autonomous healing for microservice-based apps. In: Proc. of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2015). 2015.

[9] Ma S, Liu I, Chen C, Lin J, Hsueh N. Version-based microservice analysis, monitoring, and visualization. In: Proc. of the 2019 26th Asia-Pacific Software Engineering Conf. (APSEC). 2019. 165−172.

[10] Kiss T, Kacsuk P, Kovacs J, Rakoczi B, Hajnal A, Farkas A, Gesmier G, Terstyanszky G. Micado-microservice-based cloud application-level dynamic orchestrator. Future Generation Computer Systems, 2019,94:937−946.

[11] Sampaio AR, Rubin J, Beschastnikh I, Rosa NS. Improving microservice-based applications with runtime placement adaptation. Journal of Internet Services and Applications, 2019,10:Article No.4.

[12] Aderaldo CM, Mendonça NC, Schmerl B, Garlan D. Kubow: Anarchitecture-based self-adaptation service for cloud native applications. In: Proc. of the 13th European Conf. on Software Architecture. 2019. 42−45.

[13] Gabbrielli M, Giallorenzo S, Guidi C, Mauro J, Montesi F. Self-reconfiguring Microservices. Cham: Springer Int'l Publishing, 2016. 194−210.

[14] Wang Y. Towards service discovery and autonomic version management in self-healing microservices architecture. In: Proc. of the 13th European Conf. on Software Architecture. 2019. 63−66.

[15] Akbulut A, Perros HG. Software versioning with microservices through the API gateway design pattern. In: Proc. of the 2019 9th Int'l Conf. on Advanced Computer Information Technologies (ACIT). 2019. 289−292.

[16] Paques H, Liu L, Pu C. Adaptation space: A design framework for adaptive Web services. Int'l Journal of Web Services Research (IJWSR), 2004,1(3):1−24.

[17] Guerrero C, Lera I, Juiz C. Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. Future Generation Computer Systems, 2019,97:131−144.

[18] Guerrero C, Lera I, Juiz C. A lightweight decentralized service placement policy for performance optimization in fog computing. Journal of Ambient Intelligence and Humanized Computing, 2019,10:2435−2452.

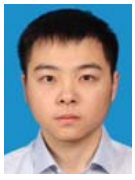
附中文参考文献:

[2] 冯志勇,徐砚伟,薛霄,陈世展.微服务技术发展的现状与展望.计算机研究与发展,2020,57(5):1103−1122.

[3] 吴化尧,邓文俊.面向微服务软件开发方法研究进展.计算机研究与发展,2020,57(3):525−541.

贺祥(1997－),男,博士生,CCF 学生会员,主要研究领域为微服务,自适应系统,边缘计算,服务计算.

涂志莹(1983－),男,博士,副教授,CCF 专业会员,主要研究领域为软件工程,服务计算,知识工程,企业业务建模技术.



刘磊(1998－),男,硕士生,主要研究领域为微服务,服务计算.

徐晓飞(1962－),男,博士,教授,博士生导师,CCF 会士,主要研究领域为服务计算与软件服务工程,大服务与务联网,企业智能计算,企业业务建模技术.