软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2017,28(6):1435−1454 [doi: 10.13328/j.cnki.jos.005223]

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

一种基于执行轨迹监测的微服务故障诊断方法*

王子勇[1,2], 王 焘[1], 张文博[1], 陈宁江[3], 左 春[1,4]

[1](中国科学院 软件研究所,北京 100190)

[2](中国科学院大学,北京 100190)

[3](广西大学 计算机与电子信息学院,广西 桂林 530004)

[4](中科软科技股份有限公司,北京 100190)

通信作者: 王焘, E-mail: wangtao@iscas.ac.cn

摘 要: 微服务正逐步成为互联网应用所采用的设计架构,如何有效检测故障并定位问题原因,是保障微服务性能与可靠性的关键技术之一.当前的方法通常监测系统度量,根据领域知识人工设定报警规则,难以自动检测故障并细粒度定位问题原因.针对该问题,提出一种基于执行轨迹监测的微服务故障诊断方法.首先,利用动态插桩监测服务组件的请求处理流,进而利用调用树对请求处理的执行轨迹进行刻画;然后,针对影响执行轨迹的系统故障,利用树编辑距离来评估请求处理的异常程度,通过分析执行轨迹差异来定位引发故障的方法调用;最后,针对性能异常,采用主成分分析抽取引起系统性能异常波动的关键方法调用.实验结果表明:该方法可以准确刻画请求处理的执行轨迹,以方法为粒度,准确定位系统故障以及性能异常的问题原因.

关键词: 故障诊断;异常检测;微服务;执行轨迹;主成分分析

中图法分类号: TP311

中文引用格式: 王子勇,王焘,张文博,陈宁江,左春.一种基于执行轨迹监测的微服务故障诊断方法.软件学报,2017,28(6):1435−1454. http://www.jos.org.cn/1000-9825/5223.htm

英文引用格式: Wang ZY, Wang T, Zhang WB, Chen NJ, Zuo C. Fault diagnosis for microservices with execution trace monitoring. Ruan Jian Xue Bao/Journal of Software, 2017,28(6):1435−1454 (in Chinese). http://www.jos.org.cn/1000-9825/5223.htm

Fault Diagnosis for Microservices with Execution Trace Monitoring

WANG Zi-Yong[1,2], WANG Tao[1], ZHANG Wen-Bo[1], CHEN Ning-Jiang[3], ZUO Chun[1,4]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100190, China)

[3](School of Computer,Electronics and Information, Guangxi University, Guilin 530004, China)

[4](Sinosoft Company Limited, Beijing 100190, China)

Abstract: Microservice architecture is gradually adopted by more and more applications. How to effectively detect and locate faults is a key technology to guarantee the performance and reliability of microservices. Current approaches typically monitor physical metrics, and manually set alarm rules according to the domain knowledge. However, these approaches cannot automatically detect faults and locate root causes in fine granularity. To address the above issues, this work proposes a fault diagnosis approach for microservices based on the

* 基金项目: 国家自然科学基金(61402450, 61363003, 61572480); 北京市自然科学基金(4154088); CCF-启明星辰"鸿雁"科研资助计划(CCF-VenustechRP2016007); 国家科技支撑计划(2015BAH55F02)

Foundation item: National Natural Science Foundation of China (61402450, 61363003, 61572480); Natural Science Foundation of Beijing(4154088); CCF-Venustech Hongyan Research Initiative (CCF-VenustechRP2016007); National Key Technology R&D Program of China under Project(2015BAH55F02)

收稿时间: 2016-07-21; 修改时间: 2016-10-11; 采用时间: 2016-12-22; jos 在线出版时间: 2017-02-20

CNKI 网络优先出版: 2017-02-20 15:14:47, http://www.cnki.net/kcms/detail/11.2560.TP.20170220.1514.032.html

execution trace monitoring. First, dynamic instrumentation is used to monitor the execution traces crossing service components, and then call trees are used to describe the execution traces of user requests. Second, for the faults affecting the structure of execution traces, the tree edit distance is used to assess the abnormality degree of processing requests, and the method calls leading to failures are located by analyzing the difference between execution traces. Third, for the performance anomalies leading to the response delay, principal component analysis is used to extract the key method invocations causing unusual fluctuations in performance metrics. Experimental results show that this new approach can accurately characterize the execution trace of processing requests, and locate the methods that cause system failures and performance anomalies.

Key words: fault diagnosis; anomaly detection; microservices; execution trace; principal component analysis

互联网应用的动态性和复杂性不断增加,传统的软件架构已经难以适应用户需求的快速变化.微服务(microservices)[1,2]将复杂的软件系统拆分成功能单一、可独立开发部署的服务组件,通过轻量级通信机制使得这些组件协同配合,从而形成一种高内聚低耦合的系统架构.SOA[3]和微服务一脉相承,微服务将服务化理念进一步精进,其不再强调 SOA 架构中重量级的服务总线,核心思想是有效拆分应用,实现敏捷开发与部署,设计与开发易维护和易扩展的分布式软件[2].但是微服务的组件众多,依赖关系复杂,软件更新频繁,增加了故障发生的概率和诊断的难度.特别是当其中一个服务组件出现故障时,故障的影响会随着模块间的相互调用不断扩散,最终导致服务质量下降或违约.因此,有效检测系统故障并准确定位问题原因,是保障微服务性能与可靠性的关键技术之一.

引发系统故障的原因有很多,比如软件设计缺陷、代码问题、配置错误等.故障会导致系统行为异常,表现为请求失败、响应延时等.当前的故障诊断方法通常监测系统度量,根据领域知识,人工设定报警规则[4].但面对微服务,由于服务间交互关系复杂,系统管理员难以设定合理的故障检测规则,并且无法细粒度定位问题原因.端到端的执行轨迹[5−8]可以记录请求在系统内部的处理过程,包括经过的服务组件、调用关系、执行时间等.由于系统出现故障通常会引起执行轨迹的偏移[5],通过对执行轨迹的分析,可快速有效定位故障原因.然而,基于执行轨迹监测的微服务故障诊断方法面临以下挑战:首先,微服务的一个请求处理需要多个相互独立的组件协同配合完成,因而难以监测与特定请求相对应的跨结点执行轨迹;其次,不同的微服务有不同的业务逻辑,同一微服务的业务种类繁多(例如,支付宝系统有几十个关键链路和几百个非关键链路),因而难以获取众多不确定的执行轨迹;最后,微服务组件通常会有多个运行实例,因此难以准确定位是哪个运行实例的哪个环节出现故障.

针对上述问题,本文提出一种基于执行轨迹监测的微服务故障诊断方法.本文首先利用跨服务组件的执行轨迹监测及约简方法对执行轨迹进行了刻画.然后,从系统错误和性能异常两方面进行了故障诊断:在系统错误诊断方面,利用调用树的编辑距离来评估请求处理的异常程度,通过对比分析执行轨迹的差异,准确定位发生错误的方法调用;在检测性能异常方面,利用主成分分析提取对响应时间延迟造成影响较大的组件实例与方法调用.本文的主要贡献包括以下几点:

(1) 提出一种面向微服务架构的可插拔、易扩展的跨组件执行轨迹监测方法以及基于调用树的执行轨迹刻画与自动构建方法;
(2) 提出一种基于树编辑距离的执行轨迹异常评估方法,并利用宽度优先搜索对比执行轨迹的问题定位方法;
(3) 提出一种基于主成分分析的性能异常检测与问题定位方法;
(4) 利用 TPC-W 基准测试,通过注入一系列典型的错误来验证了所提出方法的有效性.

本文第 1 节介绍整体思想.第 2 节给出基于执行轨迹的故障诊断方法.第 3 节利用 TPC-W 基准测实验证方法的有效性.第 4 节分析并比较相关工作.最后,第 5 节总结本文的工作,并提出下一步的工作.

1 研究思路

本文工作的目标是提出一种面向微服务的高效故障诊断方法.研究思路如图 1 所示,主要包含执行轨迹监测、执行轨迹构建和故障诊断这 3 个环节.

(1) 执行轨迹监测:本文利用动态插桩的方式,在微服务方法调用处插入监测代码,收集方法的执行信息,包括方法唯一标识、处理时间、服务组件唯一标识等.由于请求通常是由多个服务组件协同处理,本文在远程调用协议中加入了方法调用关系,以实现跨组件的请求轨迹的监测.根据以上监测信息,利用调用树对执行轨迹进行刻画;

(2) 执行轨迹构建:由于程序的分支判断等,同一服务可能存在多种不同的执行轨迹.在覆盖测试阶段与运行阶段搜集监测数据,对请求处理的执行轨迹进行刻画,从而尽可能全面地获取每类请求的所有正常执行轨迹,并保存在执行轨迹库中作为参考的基准.同时,为了消除递归、循环调用等对执行轨迹的影响,增加了相应的约简规则;

(3) 故障诊断:本文将引起执行轨迹偏离的故障分为系统错误和性能异常,表现为执行轨迹结构和执行时间的变化.前者故障通常导致了不同的执行轨迹,本文采用树的编辑距离来评估请求的异常程度,通过对比分析定位引起故障的方法调用;后者故障通常是资源竞争导致性能衰减,表现为服务响应时间延迟.由于执行轨迹通常包含上百个方法调用,本文通过主成分分析对监测数据进行降维,进而提取关键的方法调用,缩小性能定位范围.

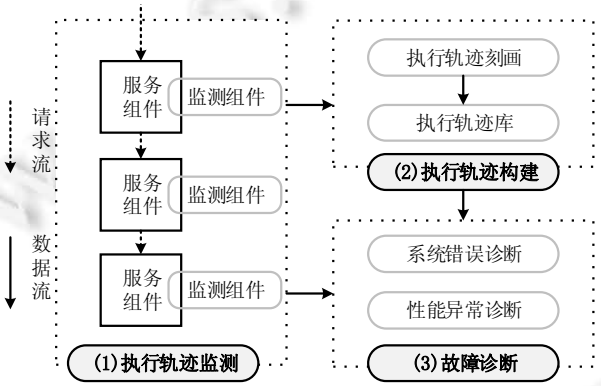

Fig.1 Overview of fault diagnosis for microservices based on execution trace

图 1 基于执行轨迹的微服务诊断方法研究思路

2 基于执行轨迹的故障诊断方法

2.1 执行轨迹监测与构建

2.1.1 执行轨迹的刻画

请求的处理是由若干服务组件协同完成,其执行轨迹是各服务组件的方法调用.本文采用方法调用树来刻画请求处理的执行轨迹,如图 2 所示.

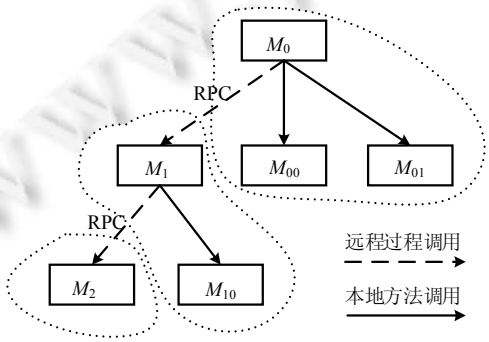

Fig.2 Call tree representation for execution traces

图 2 执行轨迹的调用树表示

调用树中的结点表示方法调用,有向边表示方法调用关系,虚线内结点表示运行于同一服务组件的方法,结点 M_i 用多元组(1)表示:

$$M_i=(requestUID,methodUID,callerUID,calleeList,info) \tag{1}$$

其中,*requestUID* 为请求标识符,在请求入口处生成,例如第 3 节实验验证采用的网上书店所提供的 14 种服务入口处;*methodUID* 为方法标识符;*callerUID* 为父方法标识符;*calleeList* 为子方法列表;*info* 包含方法的其他信息,用多元组(2)表示:

$$info=(callType,serviceUID,order,startTime,endTime,duration) \tag{2}$$

其中,*callType* 为方法的调用类型,分为本地调用和远程过程调用(remote process call,简称 RPC),如图 2 所示,M_0 远程调用 M_1,本地调用 M_{00} 和 M_{01};*serviceUID* 为方法所在服务组件的标识符,例如网上书店包括浏览服务、订单服务、结算中心等服务组件.另外,每个服务组件又包含若干运行实例.*ServiceUID* 由服务组件和实例编号唯一标识;*order* 为方法的调用顺序,子结点按照从左到右的顺序排序为方法的调用时序关系;*startTime* 和 *endTime* 是方法开始、结束时间;*duration* 为方法的执行时间,但不包括子方法的执行时间.

2.1.2 执行轨迹的监测

为满足细粒度及扩展性要求,本文采用一种动态插桩的方式获取方法调用的执行信息,采用开源 Java 字节码操控框架 ASM[9]对应用程序进行字节码修改,在虚拟机启动时,通过增加代理[10]的方式将监测代码插入到指定方法.

相对于本地应用程序,微服务的方法调用包括本地调用和远程调用,执行轨迹的监测需要解决以下难点.

(1) 多服务组件的各个请求的区分及标识

我们在服务入口处为请求生成唯一标识 *requestUID*.在调用远程方法时,调用方法将 *requestUID* 传递给被调用方法,被调用方法解析该字段,从而确定了该方法调用属于哪个请求.

(2) 服务组件间方法调用关系的确定

被调用方法维护调用方法的标识符 *callerUID*,在远程方法调用时,将本方法的标识符 *methodUID* 传递给远程方法,远程方法解析该字段,确定组件间的方法调用关系.

(3) 多服务组件方法调用顺序的确定

对于本地方法调用,通过时序关系即可确定方法的调用顺序.而对于远程方法调用,由于结点的时钟难以实现完全同步[11],在执行轨迹监测时,通过各组件的时间是不能准确地确定方法的调用顺序.因此,文本在远程调用方法时,为远程方法分配一个调用顺序 *order* 字段,从而保证了监测方法调用顺序的正确性.

综合上述分析,图 3 给出了微服务跨组件的执行轨迹监测示例,服务组件 Service 1 中的方法 M_0 远程调用服务组件 Service 2 中的方法 M_2 和 Service 4 中的方法 M_4,Service 1 将三元组(*requestUID*,*callerUID*,*order*)添加到通信协议中,Service 2 和 Service 4 解析相关字段,从而实现了跨服务组件情况下对执行轨迹的监测.

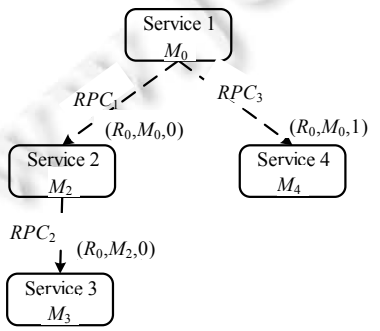

Fig.3 Example of monitoring execution traces in the microservices achitecture with multiple components

图 3 微服务跨组件的执行轨迹监测示例

各个服务组件以本地入口方法为根结点构建调用子树,确定方法的调用关系,然后根据请求标识符

requestUID 将子树汇聚,并根据方法调用关系实现执行轨迹调用树的构建.以图 2 的调用树构建为例:首先,各服务组件为每个请求构建本地调用子树,如图 4(a)~图 4(c)所示.然后,以 *requestID* 汇集各调用子树,采用自顶向下宽度优先的方法合并请求调用树,将图 4(a)和图 4(b)两个子树合并形成图 4(d),而后将图 4(c)与图 4(d)合并,形成该请求的调用树图 4(e).

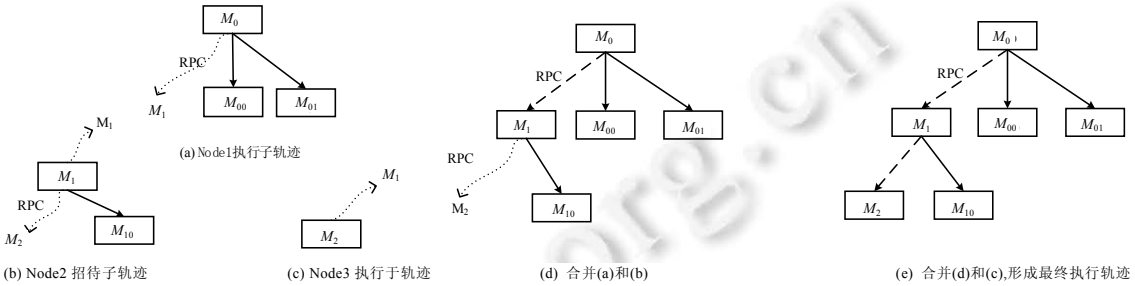

Fig.4 Example of building the call tree of execution traces

图 4 执行轨迹的调用树构建示例

需要注意的是:由于存在方法循环调用和递归调用,逻辑上等价的执行轨迹生成的调用树不同.比如,循环调用将造成调用树结构受输入参数的影响,调用树结构的宽度随循环次数的增加而变宽;同样,调用树的深度与递归的深度成正比.循环和递归调用将导致同一服务的执行轨迹种类难以确定,需要进行约简处理,因此,本文增加了约简规则及方法.

- 如果调用树中存在循环调用,则结点 $M_1,M_2,\ldots,M_n$ 有相同的方法标识,并且拥有相同的父结点;
- 如果调用树中存在递归调用,则结点 $M_1,M_2,\ldots,M_n$ 有相同的方法标识,并且结点互为父子关系;

上述规则可以确保调用树中的循环和递归可以被识别出来,进而将循环和递归中的结点汇总为一个新结点,以消除循环和递归以实现约简,其中,执行时间取结点的平均时间.

2.2 执行轨迹的构建

由于存在参数检查和分支判断,同一服务请求的处理过程可能存在多种执行轨迹.例如,本文第 3 节网上书店 Search request 服务提供了按照书名、作者和主题等 3 种查询方式,每种查询对应的执行轨迹是不同的.构建的执行轨迹集合需要具有较高的准确性及覆盖率,否则将影响故障诊断结果.覆盖测试是成熟的软件项目上线运行前必不可少的环节,我们在该阶段对软件系统的执行轨迹进行监测,以构建执行轨迹集合,将其作为检测系统故障的基准.

服务的执行轨迹集合 S 构建过程如下.

(1) 初始阶段,轨迹集合 S 为空;

(2) 针对执行轨迹 T_i,通过宽度优先搜索的树匹配算法与集合 S 中已有的轨迹 C_j 进行匹配:

- 如果匹配成功,则继续下一个执行轨迹的匹配;
- 如果匹配失败,则集合 S 新增执行轨迹 C_i;

执行轨迹集合的构建依赖于测试的覆盖率,覆盖率越高,得到的基准执行轨迹集合就越全面,则故障诊断准确率越高.如何进行覆盖测试目前已经得到了广泛的研究[12,13],但不是本文讨论的重点.为了避免覆盖测试阶段遗漏的正确执行轨迹,在软件系统上线运行阶段,管理员发现异常的执行轨迹时,可以根据经验进行修正,发现、确认新的正确执行轨迹,并将其加入基准执行轨迹集合.

2.3 故障诊断

系统故障会导致执行轨迹发生偏离,表现为执行轨迹结构的变化和执行时间的波动,本文分别将其称为结构故障和性能异常故障,并针对这两类故障分别提出相应的故障诊断方法,实现了方法粒度的故障定位.

2.3.1 结构故障诊断

同一服务请求处理的执行轨迹集合包含了其可能的执行轨迹,以此为基准,我们将运行时监测到的执行轨迹与其对比分析,从而对某一请求进行系统错误定位.本文采用树编辑距离[14]来评估执行轨迹的异常程度,对超过异常阈值的请求,实现了方法级别的故障定位.树的编辑距离是对于两棵树 T_1 和 T_2,利用编辑操作,比如增加、删除和修改,实现将 T_1 转换为 T_2 所需的操作代价.对于请求 i 的执行轨迹 T_i,与执行轨迹 C_j 的编辑距离可定义为

$$\delta(T_i,C_j)=\min\{\gamma(M)\},C_j\in T_i\text{所属服务的执行轨迹集合 }S \tag{3}$$

其中,T_i 为请求 i 的执行轨迹;C_j 为该请求所属服务的轨迹集合 S 中的一个执行轨迹;M 为 T_i 到 C_j 的编辑映射,$M\in V(T_i)\times V(C_j)$(V 表示树的结点集合)为 T_i 到 C_j 的编辑映射集合.$\gamma(M)$为 T_i 经过 M 映射后转换为 C_j 的代价,如公式(4):

$$\gamma(M)=\sum_{(\upsilon,\omega)\in M}\gamma(\upsilon\to\omega)+\sum_{\upsilon\in T_i}\gamma(\upsilon\to\lambda)+\sum_{\omega\in C_j}\gamma(\lambda\to\omega) \tag{4}$$

其中,λ为空结点,$(\upsilon\to\omega)$为结点υ修改成ω的代价,$(\upsilon\to\lambda)$为删除结点υ的代价,$(\lambda\to\omega)$为增加结点ω的代价.本文将上述 3 种操作的代价设置相等的常量.

计算树的编辑距离问题包含求其子树的编辑距离问题,并且求解问题的递归算法会反复地求解相同的子问题,满足最优子结构和子问题重合两个要素.

我们采用动态规划算法来求解,计算过程可以抽象为如下推导公式:

$$\delta(F_i,F_C)=\min\begin{cases}\delta(F_i-\upsilon,F_C)+\gamma(\upsilon\to\lambda)\\ \delta(F_i,F_C-\omega)+\gamma(\lambda\to\omega)\\ \delta(F_i(\upsilon),F_C(\omega))+\delta(F_i-T_i(\upsilon),F_C-C(\omega))+\gamma(\upsilon\to\omega)\end{cases} \tag{5}$$

其中,

$$\delta(\varnothing,\varnothing)=0 \tag{6}$$

$$\delta(F_i,\varnothing)=\delta(F_i-\upsilon,\varnothing)+\gamma(\upsilon\to\lambda) \tag{7}$$

$$\delta(\varnothing,F_C)=\delta(\varnothing,F_C-\omega)+\gamma(\lambda\to\omega) \tag{8}$$

其中,F_i,F_C 分别为请求 i 的执行轨迹 T_i 和轨迹 C 的子树构成的森林,森林之间按照子树结点表示的方法调用顺序进行排序.$\varnothing$为空树.$F_i-\upsilon$表示从 F_i 中删除顶点υ生成的森林.$F_i-T(\upsilon)$代表删除以υ为顶点的子树构成的森林.该定义为递归定义,当递归完成,即可得到执行轨迹 T_i 与基准轨迹的编辑距离.

为了确定异常发生的方法调用,需要找出最相近的基准轨迹进行对比.由于轨迹集合 S 中树的结点数量是不等的,仅按照编辑距离是不够确定轨迹 T_i 与哪个基准轨迹最相近.因此,基于树的编辑距离的定义,我们使用公式(9)来定义树的相似度:

$$Sim(T_i,C_j)=\frac{|V(T_i)|+|V(C_j)|-\delta(T_i,C_j)}{|V(T_i)|+|V(C_j)|} \tag{9}$$

其中,T_i 为请求 i 的执行轨迹,C_j 为其中一种基准轨迹,$V(T_i)$和 $V(C_j)$分别为 T_i 和 C_j 的结点数,$\delta(T_i,C_j)$为 T_i 和 C_j 的编辑距离.

进一步,当树的相似度较较低时,则异常的程度越大,我们使用公式(10)来评估执行轨迹 T_i 的异常程度(abnormality degrees,简称 AD):

$$AD=1-\max(Sim(T_i,C_j)),C_j\in T_i\text{所属服务的执行轨迹集合 }S \tag{10}$$

如果 AD 大于预置的阈值γ,则表示该请求发生了错误.阈值的选取将影响故障诊断结果:如果阈值设置过大,则容易造成诊断遗漏;如果设置过小,则将导致误报率增加.本文第 3 节实验验证部分,依据领域经验设置γ为 0.15,并且根据后续监测结果进行调整.

由于该请求与轨迹 C 具有最大的相似度,通过比较 T_i 和 C 的轨迹差别,可以定位到错误出现在具体哪个方法,方法粒度的错误定位采用算法 1.

算法 1. 宽度优先的错误方法调用定位算法.

Input:T_i:请求 i 的执行轨迹;C_j:与 T_i 相似度最大的基准轨迹;

Output:*method*:错误发生处的方法调用.

Pseudocode:

```
1.  create empty queue Qc, Qi
2.  Qc.enqueue(Cj.root)
3.  Qi.enqueue(Ti.root)
4.  Set methodc, methodi
5.  while Qi is not empty:
6.      methodi=Qi.dequeue()
7.      if Qc is empty:
8.          return methodi
9.      end if
10.     methodc=Qc.dequeue()
11.     if methodc.UID is not equal to methodi.UID:      //如果方法标识不一致,则此处发生错误
12.         reuturn methodi
13.     end if
14.     for all calleei in methodi.calleec do:           //将 methodi 子方法加入队列 Qj
15.         Qi.enqueue(calleei)
16.     end for
17.     for all calleec in methodc.calleeList do:        //将 methodc 子方法加入队列 Qc
18.         Qc.enqueue(calleec)
19.     end for
20. If methodc.calleeList is empty:                      //轨迹中剩余方法方法调用
21.     return methodi
22. end if
```

针对异常轨迹 T_i,在计算异常程度 *AD* 时,我们可以得到与 T_i 最相似的基准轨迹 C_j,通过与 C_j 的匹配,我们就可以定位异常发生所在的方法调用,轨迹中包含了服务组件 *serviceUID* 等信息,因而可以进一步确定异常方法所在的服务组件.算法 1 采用宽度优先算法进行故障定位,依次将 C_j 和 T_i 的根结点入队列(第 1 行~第 3 行);然后,循环进行方法对比(第 5 行~第 19 行),当方法标识不匹配时,即确定了错误发生的位置(第 11 行).此外,T_i 可能会包含错误的方法调用,算法结尾处增加了相应的判断(第 20 行~第 22 行).其中,算法 1 需要存储轨迹中的结点信息,其空间复杂度为 $O(|T_i|+|C_j|)$,其中,$|T_i|$和$|C_j|$分别表示请求 i 的轨迹 T_i 和轨迹 C_j 的结点数;在定位故障方法调用时,需要循环对比,时间复杂度为 $O(|T_i|\times|C_j|)$.

2.3.2 性能异常故障诊断

同一执行轨迹有相同方法调用树,例如网上书店应用 Search request 服务,按照书名进行查询.此外,执行时间也相对稳定[6,7],如果执行时间出现大幅度波动,则该请求处理出现性能异常.我们采用变异系数(coefficient of variation,简称 CV)来衡量执行时间的异常程度:

$$CV=\frac{\sigma}{\mu}\times 100\% \tag{11}$$

其中,

$$\sigma=\sqrt{\frac{1}{n-1}\sum_{i=0}^{n}(x_i-\mu)^2} \tag{12}$$

$$\mu=\frac{1}{n}\sum_{i=0}^{n}x_i \tag{13}$$

其中,x_i为第 i 个请求的执行时间,μ为某类请求的平均执行时间,σ为标准差,CV 为标准差与均值的比值.变异系数 CV 较大,则表明系统在处理请求时响应时间波动幅度较大,性能异常程度较高,则需要进行性能分析.一个执行轨迹往往包含上百个方法调用,例如第 3 节网上书店 Buy Confirm 服务某一轨迹包含了 57 个方法调用.方法间存在调用关系,包含了冗余数据,需要从中选取引起性能异常的关键方法,以缩小异常定位范围.

在多元统计分析中,当超过 10 个变量时,由于被丢弃的变量往往是冗余的,选取其中的一个子集就可以不改变最终分析结果[15].主成分分析(principle component analysis,简称 PCA)[16]是一种常用的多元分析方法,将 n 维数据削减为 p 个主成分($p<n$)以表现原始数据信息.在性能异常故障诊断时,由于某些方法间存在较强的关联关系,因此利用 PCA 可有效降低原始数据的维度,从而缩小问题定位的范围.

基于 PCA 的性能异常故障诊断步骤如下.

(1) 构建请求处理矩阵

PCA 的输入为矩阵,首先需要将执行轨迹转换为执行序列.本文采用调用树表示执行轨迹,树结点满足一定父子关系和时序关系,因此可将调用树转换为等语义的执行序列.我们利用基于时间序列的深度优先搜索算法,将调用树 T 转换为执行序列.各个请求的执行序列按行排列,组成 PCA 分析的输入矩阵 A:

$$A=\begin{bmatrix} t_{11} & t_{12} & \cdots & t_{1n} \\ t_{21} & t_{22} & \cdots & t_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ t_{m1} & t_{m2} & \cdots & t_{mn} \end{bmatrix} \tag{14}$$

其中,m 为请求数量,n 为执行轨迹的方法数量.列表示方法的执行时间,即,t_{ij} 为请求 i 执行轨迹中方法调用 M_j 的执行时间.

(2) 主成分分析

执行轨迹序列中,每个方法的执行时间是不同的,需对原始输入矩阵 A 进行标准化转换,得到标准化矩阵 Z:

$$Z_{ij}=\frac{t_{ij}-\overline{\mu}_j}{\sigma_j}, i=1,2,...,m; j=1,2,...,n \tag{15}$$

其中,

$$\overline{\mu}_j=\frac{1}{m}\sum_{i=0}^{m}t_{ij}, j=1,...,n \tag{16}$$

$$\sigma_j=\sqrt{\frac{1}{m-1}\sum_{i=0}^{m}(t_{ij}-\overline{\mu}_j)^2}, j=1,...,n \tag{17}$$

然后,求标准化矩阵 Z 的协方差矩阵Σ:

$$\Sigma=\begin{bmatrix} \rho_{11} & \rho_{12} & \cdots & \rho_{1n} \\ \rho_{21} & \rho_{22} & \cdots & \rho_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n1} & \rho_{n2} & \cdots & \rho_{nn} \end{bmatrix} \tag{18}$$

其中,

$$\rho_{ij}=\frac{cov(Z_i,Z_j)}{\sqrt{DZ_i}\sqrt{DZ_j}}, cov(Z_i,Z_j)=E((Z_i-E(Z_i))\cdot(Z_j-E(Z_j))) \tag{19}$$

最后,求协方差矩阵Σ的特征值和特征向量,求解特征方程(20):

$$\Sigma X=\lambda X \tag{20}$$

得到特征值$\lambda_1,\lambda_2,...,\lambda_n$及对应的特征向量$\boldsymbol{\mu}_1,\boldsymbol{\mu}_2,\ldots,\boldsymbol{\mu}_n$.

(3) 主成分选取

主成分的选取决定了数据压缩率,设选取的主成分个数为 k.特别地,如果 $k=n$,相当于在原始数据上进行了转换,保留了原始数据的100%信息.那么在选择 k 值时,以 k 个主成分可以保留的方差百分比为参考依据,百分比越大,选取的主成分所表示的信息,与原始数据越近似.首先对特征值$\lambda_1,\lambda_2,...,\lambda_n$ 按照降序排序,然后计算主成分方差百分比,如公式(21):

$$\frac{\sum_{j=1}^{k}\lambda_j}{\sum_{i=1}^{n}\lambda_i}\geqslant\beta \tag{21}$$

根据文献[16]的建议,我们设置β为 0.85.

(4) 异常方法的定位

主成分实际上是原有维度的线性组合,而系数向量就是对应的特征向量,如公式(22):

$$\begin{aligned} p_1 &= a_{11}t_1 + a_{12}t_2 + ... + a_{1n}t_n \\ p_2 &= a_{21}t_1 + a_{22}t_2 + ... + a_{2n}t_n \\ &... \\ p_m &= a_{m1}t_1 + a_{m2}t_2 + ... + a_{mn}t_n \end{aligned} \tag{22}$$

其中,p_i 表示主成分 i;变量 t_i 表示方法 M_i 执行时间;a_{ij} 表示主成分 p_i 对于变量 t_i 的系数,表示了主成分与原始数据维度的相关性,系数越大,则表示该维度对主成分贡献越大[15],即,该维度对应的方法即为引起性能问题的主要因素.于是,我们给出性能异常故障的定位算法,见算法 2.

算法 2. 性能异常故障定位算法.

Input:*pList*:选取的 k 个主成分列表,包含特征值λ及特征向量μ;*traceList*:执行轨迹列表;*C*:对应的执行轨迹;

Output:*perfMethodList*:性能异常方法列表.

Pseudocode:

```
1.  Set methodSet={}, perfMethodList={}
2.  sortedpList=sortByEigenValue(pList,DESC)          //对 k 个主成分按照特征值降序排序
3.  for all p in sortedpList do:
4.    maxCoeff,method=max(p.μ,C)                      //获得最大系数及对应方法
5.    factor=maxCoeff*p.λ                             //计算该方法的权重因子
6.    methodSet.add({method,factor})
7.  end for
8.  sortedMethodList=sortByFactor(methodSet,DESC)     //对选取的 k 个方法按照因子降序排序
9.  for all method in sortedMethodList do:
10.   serviceTraceList=aggregateBySUID(traceList,method)   //按照服务组件 ID 聚集
11.   Set serviceTimeSet={}                           //组件指定方法的执行时间集合
12.   for all serviceTraces in tmpTraceList do:
13.     serviceUID,avgTime=avg(tmpTraceList,method)   //求组件的平均时间
14.     serviceTimeSet.add({serviceUID,avgTime})
15.   end for
16.   serviceTimeList=sortByTime(serviceTimeSet,DESC) //以 avgTime 进行降序
17.   perfMethodList.add(method,serviceTimeList)      //添加到输出列表中
18. end for
19. return perfMethodList
```

算法 2 首先对选取的 k 个主成分 $p_1,p_2,\ldots,p_k$,按照对应的特征值$\lambda_1,\lambda_2,...,\lambda_k$ 按照从大到小排序(第 2 行),排序越靠前,该主成分愈显著;然后,依次选取主成分最大系数及对应的方法,并计算方法的权重因子,得到 k 个方法及权重因子(第 3 行~第 7 行);其次,对选取的 k 个方法按照权重因子进行倒序排序(第 8 行);然后,针对每个方法

计算在每个服务组件的平均执行时间,并且按照平均执行时间进行倒序排序(第 9 行~第 18 行);最后的输出结果包含了选取的 k 个方法及对应的在各个服务组件的执行时间(第 19 行);其中,选取的 k 个方法调用即为引发性能异常的关键环节.更进一步,管理员根据每个方法在每个服务组件的执行时间就可以确定性能异常是由于哪个服务组件的哪个方法引起的.其中,算法 2 需要申请数据集合存储中间结果和返回结果,因此,其空间复杂度为 $O(m(C))$,其中, $m(C)$表示轨迹 C 包含的方法调用数量;在定位故障方法时,首先需要对 k 个主成分进行排序,时间复杂度为 $O(k\log k)$,其次,循环计算权重,时间复杂度为 $O(k)$;最后,主循环的时间复杂度为 $O(m(C)\times(T_s+S_s+S_s\log S_s))$,其中,$T_s$ 表示监测轨迹数,S_s 表示服务组件数,因此,算法 2 的时间复杂度为 $O(2k\log k+k+m(C)\times(T_s+S_s+S_s\log S_s))$.

3 实验及结果分析

3.1 实验环境

为验证方法的有效性,本文选取了中国科学院软件研究所自主研发的符合 TPC-W 规范[17]的基准测试套件 Bench4Q[18].TPC-W 是非盈利组织 TPC[19]提出的事务型网络服务基准,对数据模型、业务模型、性能度量、负载方法、扩展性等进行了规范.其通过模拟用户请求产生负载,以对服务及环境的性能进行评估,在学术界和工业界得到广泛认可和采用[20−23],具有很好的典型性和可重现性.Bench4Q 在遵守 TPC-W 规范的基础上,提出了一种面向服务质量的电子商务测试基准,它在模拟负载仿真、度量分析等多个方面对 TPC-W 进行了扩展.

根据相关文献[2,24−26],微服务显著的特征主要如下:(1) 服务单一职责原则,高内聚低耦合;(2) 采用轻量级通信方式,对外提供服务接口,支持异构、多技术栈实现;(3) 支持独立部署及升级等.本文基于微服务的特征,参考了微服务最佳实践[2,24,27],对 Bench4Q 提供的服务进行了封装与重构,系统架构如图 5 所示.

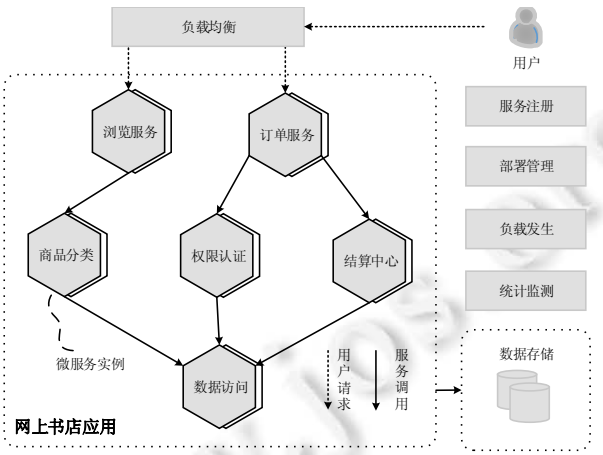

Fig.5 Bench4Q system architecture

图 5 Bench4Q 系统架构图

- 对特征(1),网上书店应用以业务模块为粒度进行服务划分,包括浏览服务、订单服务、权限认证、支付中心、商品分类和数据访问等多个服务组件;各组件职能单一,高内聚;
- 对特征(2),服务组件间通过二进制通信协议对外提供服务,服务间耦合度低,且保证了较好的扩展性;
- 对特征(3),每个服务相对较小,保证了服务质量的同时,可独立开发.

本文采用的物理实验架构如图 6 所示.Console 为管理组件,提供了测试参数配置、错误注入及监控等功能;Bench4Q 的 Agent 接受 Console 发送来的指令,模拟用户行为,访问网上书店服务;图 5 中,网上书店应用各微服务组件在应用服务器 A 和 B 各部署一个实例;负载均衡器 Nginx 为应用服务器集群提供负载均衡,采用轮询负

载策略;MySQL 数据库提供存储服务;故障诊断系统为本文所提出方法的实现.实验中,数据库、负载均衡器和应用服务器均采用默认配置,Bench4Q 设置 10 000 件商品和 1 440 000 个用户.各组件的软硬件信息见表 1.

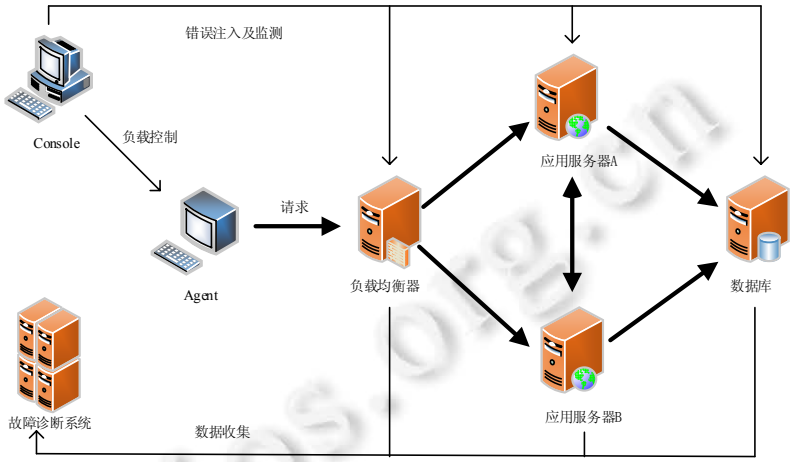

Fig.6 Experiment environment

图 6 实验环境

Table 1 Software and hardware information of experiment components

表 1 各组件软硬件信息

组件	硬件信息	软件信息	数量
console	Intel Core i5-2300/8GB RAM/500GB Disk/100Mbps NET	Bench4Q-1.3.0	1
agent	Intel Core i5-2300 CPU/4GB RAM/200GB Disk/100Mbps NET	Bench4Q-1.3.0	3
负载均衡器	Intel Xeon E5-2620 CPU/8GB RAM/500GB Disk/100Mbps NET	Nginx-1.9.13	1
应用服务器	Intel Xeon E5-2620 CPU/8GB RAM/500GB Disk/100Mbps NET	Apache-tomcat-7.0.63	2
数据库	Intel Xeon E5-2620/8GB RAM/500GB Disk/100Mbps NET	Mysql-5.6.27	1
故障诊断系统	Intel Core i5-2300/8GB RAM/200GB Disk/100Mbps NET	V0.0.1	1

3.2 执行轨迹构建

Bench4Q 网上书店应用提供了 14 种服务,本实验选取其中典型的 4 种服务作为研究对象,即 Search request(查找),Product detail(浏览),Buy Request(购买),Buy Confirm(付款).本文针对上述 4 种服务进行执行轨迹的构建,得到 4 种服务执行轨迹数如图 7 所示.

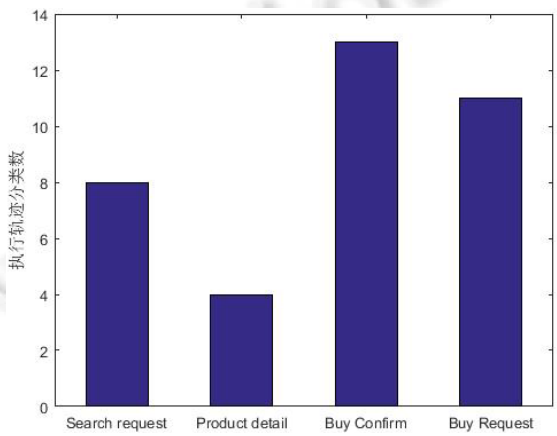

Fig.7 Execution trace number of 4 service interfaces

图 7 4 种服务构建的执行轨迹数

从图中可以发现:选取的服务执行轨迹数都在 15 个以内,Product detail 服务逻辑简单,执行轨迹数较少.我们对这 4 种服务的执行轨迹进行人工分析确认,将分析结果与方法自动生成的执行轨迹进行对比,两者结果一致.因此,我们的方法能够准确地自动刻画出服务的执行轨迹.

3.3 故障诊断

本文参照已有工作,采用注入故障的方式来验证所提方法的有效性,如文献[20,28−31]等.在注入故障列表方面,本文对众多真实软件故障及相关文献[8,32−35]涉及到的典型故障进行了分析汇总,选取了其中常见的操作系统、应用服务器、数据库和应用程序等 4 类错误进行实验,见表 2.在实验过程中,我们将错误单独注入到系统,并在Console设置每次负载持续时间为90s,并发数为100;同时,故障诊断系统收集系统的执行信息,并进行故障诊断.

Table 2 List of injected faluts

表 2 注入故障列表

故障类型	故障说明	注入数量
操作系统类	通过 Stress,TC 等压力工具模拟 CPU、内存、IO、网络等异常	10
应用服务器类	模拟 tomcat 的引起的故障,包括 JVM 配置、线程池配置、数据库连接、进程、已知容器 Bug 等	10
数据库类	数据库类故障,模拟连接数、表死锁、索引、权限等故障	10
应用程序类	应用程序本身的故障,模拟 Bug、配置错误等	10

本文选取其中 3 个典型的故障为例介绍本文提出的故障诊断方法.

- 故障(1):通过 TC 工具造成应用服务器 *A* 网络丢包 15%;
- 故障(2):设置应用服务器 *A* 数据库连接数 maxActive 为 10;
- 故障(3):使用 SELECT…FOR UPDATE 语句给订单表加一个排它锁.

对于偶然性故障,比如模拟 CPU、网络异常等,是可以恢复的,本文在第 30s 注入,持续 30s,然后恢复正常.对于持久性故障,比如 JVM 配置、数据库连接等,需要重启服务器才能恢复,因此,此类故障故障持续时间 90s.故障(1)和故障(3)在第 30s 注入,持续 30s,然后恢复正常,而故障(2)持续时间 90s.

3.3.1 系统错误的故障诊断

当注入故障后,请求的执行轨迹发生了变化,4 种服务在注入 3 个故障后的系统异常程度变化如图 8 所示.

我们选取与监测到的异常执行轨迹具有最大相似度的正常执行轨迹进行对比,即可定位出异常发生所在的方法调用及服务运行实例.

在第 30s 注入故障(1)后,4 种服务均发生请求失败,出现了服务违约.从图 8(a)~图 8(d)可知,Search request, Product detail,Buy Request 和 Buy Confirm 这 4 种服务中错误执行轨迹的异常程度分别超过了 0.16,0.41,0.38 和 0.23.通过第 2.3.1 节的算法 1,我们得到发生错误的方法均与网络相关,例如 *Database.getConnection*(), *httpConnection.readLine*(),*SocketInputStream.read*()等.然后,算法 1 定位出的错误方法以 *ServiceUID* 来统计,我们发现,发生在应用服务器 *A* 的方法占 95%以上.因此进一步,我们可以断定故障发生的位置为服务器 *A* 的网络.

注入故障(3)后,Buy Request 和 Buy Confirm 服务失效,从图 8(c)和图 8(d)可知:Buy Request 和 Buy Confirm 的执行轨迹异常程度较大,分别超过了 0.63 和 0.61.采用通过错误故障诊断方法,定位出异常发生所在的位置为新增订单 *createOrder*()和修改订单 *updateOrder*()方法,两个方法都是修改订单表,而查询订单操作并未失败,这样有效地缩小了故障排查范围.

对于故障(2),我们有意将应用服务器 *A* 数据访问服务组件的数据库连接数设置过小,即 10 个.在 100 个并发数下,在图 7 中我们可以发现:只有少数执行轨迹结构发生变化,但服务响应时间延迟较大.为进一步确定问题发生在哪个环节,在第 3.3.2 节,我们将以 Buy Confirm 服务为例进一步进行性能异常诊断.

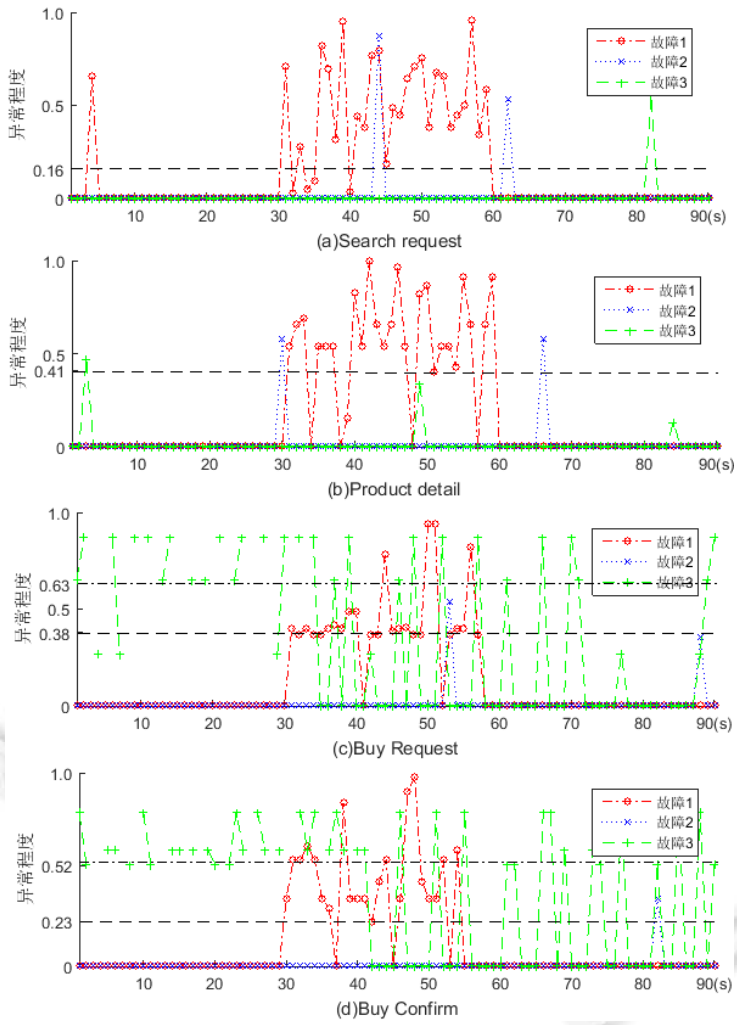

Fig.8 Abnormality degrees of the four service interfaces

图 8 4 种服务的异常程度

3.3.2 性能异常的故障诊断

Buy Confirm 服务共有 13 种执行轨迹,注入故障(2)的 90s 时间段内,各执行轨迹响应时间的变异程度 CV 如图 9 所示.本文选取变异系数最大的轨迹 5 作为分析对象,由图 10 可知:服务的响应时间波动很大,从最低 50ms 到最高 1600ms.于此同时,CPU、内存等物理资源使用率维持在 15%和 40%以下,仅依据资源利用率难以定位出问题的原因.

执行轨迹 5 包含了 57 个方法调用,每种方法作为一个维度,通过性能异常诊断,得到主成分占比如图 11 所示,主成分 1~主成分 3 分别占 64.9389%,26.2608%和 4.1277%,主成分 1 和主成分 2 累计占 91.1997%,主成分 1 和主成分 2 可以有效地表现原有数据信息.

表 3 列出了前 3 个主成分与其中 6 个方法调用的系数,系数越大,则主成分与方法调用的相关性越强.从表 3 的分析结果可以定位引起性能瓶颈的主要为 *Database.getConnection*()方法调用.进一步地,统计得到该方法调用在应用服务器 *A* 和 *B* 此方法的平均执行时间分别为 501.46ms 和 1.75ms,从中可以确定性能瓶颈发生在应用服务器 *A* 的创建数据库连接的方法调用.

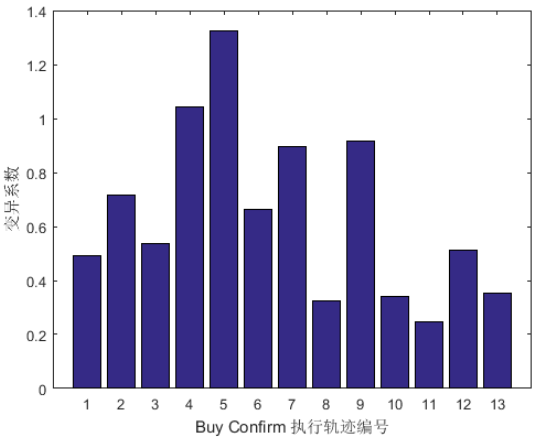

Fig.9 CV of the 13 execution traces of buy confirm service

图 9 Buy Confirm 服务 13 种执行轨迹变异系数

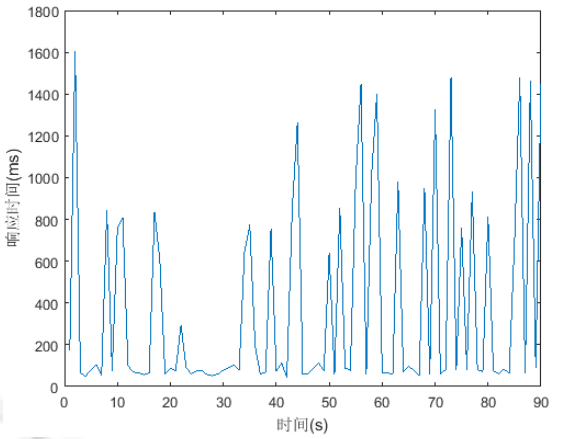

Fig.10 Response time of the 5th execution trace of buy confirm service

图 10 Buy Confirm 服务轨迹 5 响应时间

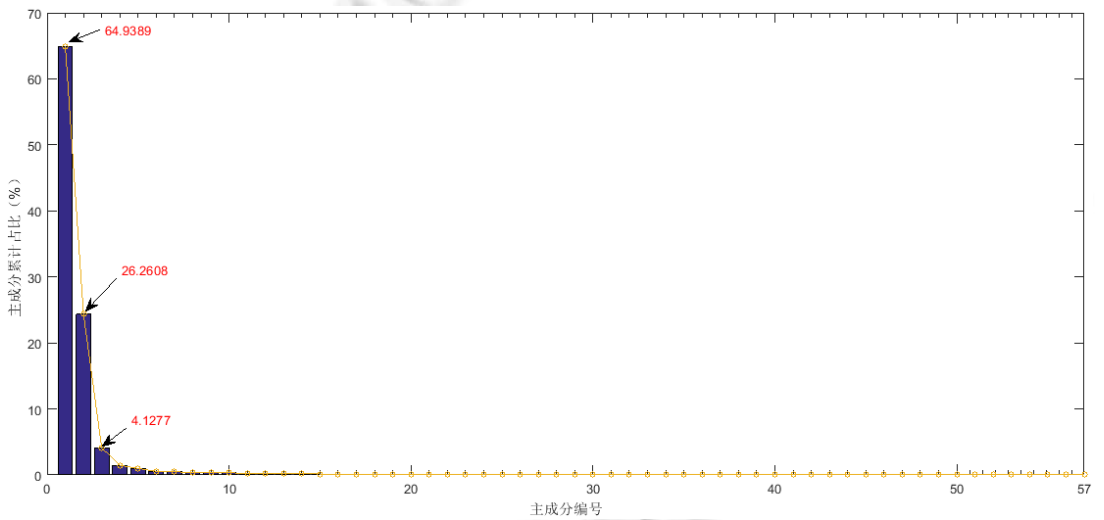

Fig.11 Ratio of the principle components

图 11 主成分占比

Table 3 Coefficient of the principle components and methods

表 3 主成分与方法的系数

method	PC1	PC2	PC3
buy_confirm_servlet.doGet	−0.000 34	0.001 025	−8.407 221
Database.getCAddr	0.000 289 4	0.003 459 1	−0.013 669
Database.doBuyConfirm	−0.001 060	−0.012 116	−0.022 504
Database.getConnection	**0.997 979**	**0.0624 943**	0.004 337
ConnectionImpl.setAutoCommit	−0.000 614	−0.002 577	0.006 646
Database.getCDiscount	−0.002 973	−0.003 538 1	0.000 708

3.3.3 实验结果

本文采用查准率(precision)(公式(23))、查全率(recall)(公式(24))、F_1 值(公式(25))分别对表 2 中注入错误的故障诊断结果进行评价.

$$precision = \frac{TP}{TP + FP} \tag{23}$$

$$recall = \frac{TP}{TP + FN} \tag{24}$$

$$F_1 = \frac{2 \times precision \times recall}{precision + recall} \tag{25}$$

其中,TP(true positive)表示能够有效定位的故障数,FP(false positive)表示误判的故障数.实验的最终结果见表 4.

Table 4 Result of diagnosis

表 4 诊断结果

故障类型	Precision	Recall	F_1
操作系统类	0.75	0.60	0.67
应用服务器类	0.89	0.80	0.84
数据库类	0.64	0.70	0.67
应用程序类	0.81	0.90	0.85
Total	0.77	0.75	0.76

实验结果表明:虽然本文提出的方法能够准确检测并定位大多数的系统故障,但仍有一些故障不能有效检测和诊断.经过分析,未能有效诊断的故障与物理资源相关,比如 CPU 利用率增加、内存使用增加等.这是由于本文提出的方法关注于执行轨迹结构和性能表现,但未出现资源瓶颈前,此类故障并不会造成请求处理错误或性能衰减.因此,可以结合物理资源的运行监测分析来及早发现此类故障.

3.4 监测性能开销及算法复杂度分析

代码插桩的监测方式会对原有系统产生一定的性能影响,本文对选取的 4 种服务插桩前后的平均响应时间通过实验进行了对比分析.实验结果如图 12 所示.在并发数为 100 的情况下,Search request,Product detail,Buy Request 和 Buy Confirm 这 4 种服务插桩前平均响应时间分别为 8.9ms,8.5ms,46.3ms 和 80.6ms,监测的方法调用数分别为 27,18,32 和 57,插桩后为 9.2ms,8.7ms,48.1ms 和 85.4ms.插桩后平均响应时间增加了约 3%~6%.同时也可发现:插桩造成的性能损耗与监测方法成正比关系,执行轨迹越复杂,监测的方法越多,性能损耗越大.其中,损耗最大的 Buy Conifirm 监测了 57 个方法调用,每个方法调用贡献了约 0.105%的性能损耗.

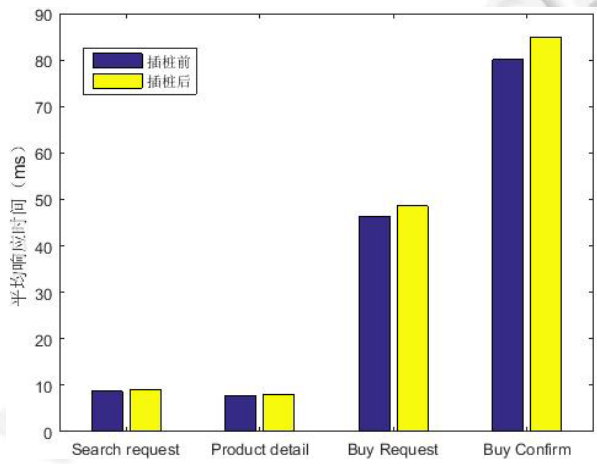

Fig.12 Average response time comparison between before and after instrumentation of the 4 services

图 12 4 种服务插桩前后平均响应时间对比

另外,第 2.3.1 节基于树相似度的错误诊断的关键是求树编辑距离,其算法的时间和空间复杂度分别是 $O(|T_1|^2|T_2|\log|T_2|)$和 $O(|T_1|^2|T_2|\log|T_2|)$[14],其中,$|T|$表示树的结点数.错误诊断需要将请求的执行轨迹与对应的分类

进行对比,假设执行轨迹数为 n,则错误诊断的时间复杂度为 $n\times O(|T_1|^2|T_2|\log|T_2|)$.为减少不必要的计算,我们在求最小相似度时,会优先于权重较大的分类进行对比,如果匹配,则不进行后续的计算.分类的权重通过请求的占比进行刻画,本文针对每个服务的执行轨迹分类所占请求的占比进行了统计.如图 13 所示,前 30%的执行轨迹约占 80%以上的请求.在实际服务功能设计时,常遵循单一职能、接口隔离的原则,因此,服务的执行轨迹数不会太大.此外,第 2.3.2 节的性能故障诊断采用了 PCA 主成分分析方法,其时间复杂度为 $O(min(p^3,n^3))$[36],其中,p 为特征数量,n 为输入矩阵的维度.PCA 分析需要存储协方差矩阵、特征值和特征向量,其空间复杂度为 $O(n^2+p+p\times n)$.

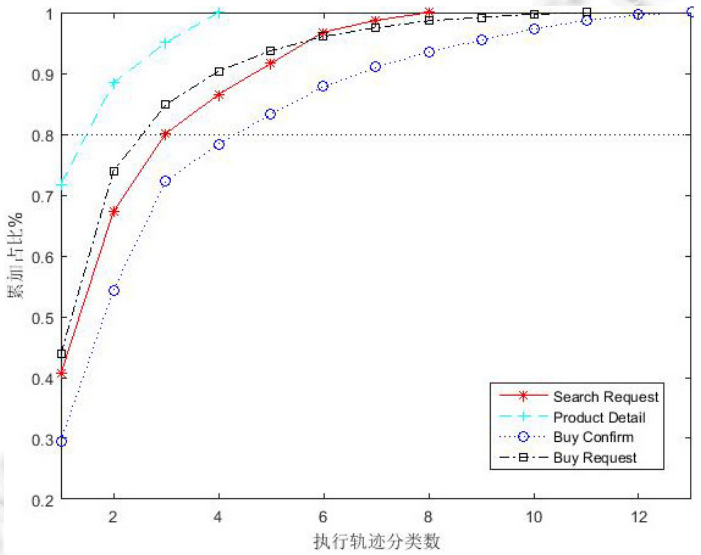

Fig.13 Cumulative ratio of the execution traces of 4 services

图 13 4 种服务的执行轨迹占比累加图

4 相关工作

由于微服务在开发、部署、管理等方面的优势,近两年得到了工业界和学术界越来越多的关注.其中,监测及故障诊断作为微服务不可或缺的一环,也得到了广泛的研究.Kubernetes[37]是构建在 Docker 之上的容器集群管理系统,简化了微服务的管理,在监测诊断方面,实现了运行度量收集、服务发现、服务自愈、失败重试等功能.Netflix Hystix[38]是一个延迟与容错库,依据对微服务运行度量和配置的实时监控进行告警与决策,来防止连锁故障的发生,保证了服务的可靠性.Twitter Fintop[39]是用于收集和打印 Fingale 服务运行信息的命令行程序,方便管理员了解微服务的状态.App-Bisect[40]通过服务版本依赖图来自动定位并修复微服务修改发布过程中的 bug 和性能恶化.Gremlin[41]是一种用于测试微服务故障处理能力的系统工具,对目标系统注入故障,通过收集到的日志信息来触发预置的故障处理动作.上述方法通常是基于度量的监控方式,能够发现并定位服务级别的故障,具体问题发生在哪个环节仍然需要投入大量成本.本文通过监测并分析请求处理的执行路径以及组件的请求处理时间,能够细粒度准确定位引发故障的服务组件及方法调用.

执行轨迹刻画了请求在系统中的处理过程,当系统出现故障时,则会直接引起执行轨迹的偏移.执行轨迹可以有帮助管理员把握系统行为特征,进而可以进行有效的故障诊断.目前,在运行监测、系统行为建模、故障诊断等方面已得到广泛地研究.

在运行监测方面,Project5[42]通过 RPC 消息时序关系和信号量处理技术构造分布式系统的执行轨迹,无需对应用、中间件或者消息进行修改,降低了调试、性能诊断的技术要求.WAP5[43]对通信消息利用因果模型和聚类分析等方法推导出消息间的关系,构造系统组件的因果关系的行为模式,进而方便性能调试.Sherlock[44]提出一种推理图模型,推断系统的关键属性.它能很好地适应根源于条件引起部分服务的退化和硬故障,利用结果可

自动检测和定位问题.Project5,WAP5 和 Sherlock 是黑盒监测系统,能够实现应用级的透明.但黑盒的缺点是检测结果一定程度上不够精确,且在统计推断关键路径时有较高的计算复杂度.文献[8,35]通过挖掘各组件的系统日志来构建每个请求的执行轨迹,这种方式针对特定系统,诊断结果受日志影响,依赖于日志质量,精度往往不高,也不具有较好的通用性.Pip[45]提供了一种声明式语言,程序员可用来记录系统的通信数据、时序和资源消耗,然后比较实际行为与预期行为的差异,发现分布式系统的结构类故障和性能问题.WebMon[46]利用异构侵入检测器和基于 Cookie 的关联器来获得精确的监测数据,实现了一种面向事务的 Web Service 性能监测系统,可实现业务、事务和系统多个维度的性能分析.Pip 和 WebMon 是基于注解来获取较精确的执行信息,依赖于应用级别的注解.X-Trace[47]对通信协议和系统中间件进行了修改,从而获得精确的网络系统的信息,实现跨组件的轨迹监测.Magpie[48]对中间件进行修改,通过核对详细的跨主机的轨迹,抽取请求的轨迹信息来对服务进行建模,进而对性能进行分析.X-Trace 和 Magpie 为了获得详细的执行信息,对库或者中间件进行了定制修改,扩展能力较弱.本文在上述监测方法的基础上,结合微服务的特点,提出一种利用动态插桩的监测方法,可以有效收集关键方法的详细执行轨迹信息,具有无需修改应用、可插拔、易扩展的优点.

在系统行为建模方面,文献[49]提出一种构建简明负载模型的机制,首先用事件模式表达系统行为,然后,通过聚类分析确定的规范请求形式.这组请求,连同它们的相对频率,是一个紧凑的工作负载模型,然后可以用于性能分析的目的.Stardust[50]记录请求经过每个服务器或者网络的活动记录,进而可监控请求进行端到端的痕迹,最终提取每一个工作负载、资源需求信息和每个资源负载延时图,细粒度的跟踪信息可以有效地离线分析系统行为.Pip[45]利用任务、消息和通知等来预先定义应用的行为模型,构造请求处理的轨迹.综合来讲,文献[45,49,50]利用预置的模型来对执行轨迹进行定义,这种方式需要开发人员有较强的领域知识,且需要投入较大成本.The Mystery Machine[51]采用依赖模型生成和关键路径计算,经过多次迭代来构造请求的执行轨迹.文献[51]所采用方法在表示系统行为模型时有较低的准确率,造成后续的分析困难;同时,挖掘算法需要多次的迭代,有较高复杂度.微服务各组件相互独立开发部署,对外开放服务接口,因此在构建请求的执行轨迹时有一定的难度.本文结合上述方法的优缺点,针对微服务架构,提出一种利用覆盖测试来构建请求的执行轨迹的方法,各组件只需要在目标方法调用处动态插入监测代码,无需理解整个系统行为,即可构建服务的执行轨迹.

在故障诊断方面,Dapper[5]是 Google 生产环境下的分布式系统跟踪平台,被用于发现系统问题,但它更通常用于探查性能不足,以及提高全面大规模的工作负载下的系统行为的理解.如何发现错误的系统行为和如何进行性能分析,文章中并未进行详细讨论.Pivot Tracing[52]提出一种功能强大的类 SQL 的查询语言,通过聚集、过滤等操作来查询执行的信息,但该方法并不能实现方法粒度的故障诊断,也不具备自动故障诊断的能力.文献[53]基于任务签名和执行时间来训练诊断分类器,在运行过程当中,通过分类器发现异常的任务,通过统计方法 *t*-test 来发现性能异常.其中.分类器的训练是关键环节,是一种监督的学习方法,构造过程有到一定的难度;同时,不同的服务处理过程有时相差很大,这时就需要构造多个分类器.Spectroscope[7]将非问题和问题两个时期的请求流图汇总,利用 Kolmogorov-Smirnov two-sample 非参数假设检验区分响应时间和结构变化的分类,然后计算每个分类对异常的贡献值进行排序,管理员根据排序列表进行故障分析.DARC[54]通过统计分析方法分析程序的调用图,以发现引发问题的根本原因.Spectroscope 和 DARC 面对的方法调用树方法数量少,没有对方法的重要程度进行区分,当面对复杂微服务时,其分析结果并未有效地缩小故障范围.Pinpoint[6]和 Pip[45]分析的粒度是模块级别或者服务器级别的,管理员仍然需要进一步排查问题的原因.本文针对执行轨迹的特点,结合非监督的分析方法实现了影响轨迹结构和性能故障的诊断,能够精确地定位到发生故障的方法及服务实例,极大地缩小了故障诊断的范围,为保障微服务的可靠性提供了较高价值的参考信息.

5 总结和下一步的工作

本文面向微服务提出一种基于执行轨迹的故障诊断方法.首先,利用动态插桩技术监测服务组件处理流程,并利用调用树对请求处理的执行轨迹进行刻画;然后,针对影响执行轨迹的故障,利用树编辑距离来评估请求处理的异常程度,通过分析执行轨迹差异来定位引发故障的方法调用;最后,针对系统性能异常,采用主成分分析

抽取引起系统性能异常波动的关键方法调用.

目前,该方法还存在如下待改进的问题:首先,动态代码注入的方式可能会造成关键方法的遗漏,后续工作将静态程序分析方法引入到插桩点定位;其次,监测方法的性能开销与插桩方法数量成正比,过多插桩会对复杂服务产生较大的性能开销,我们下一步工作将对插桩点的选取及采样频率进行研究,以较小监测开销获得较高的故障诊断准确性.

References:

[1] Namiot D, Sneps-Sneppe M. On micro-services architecture. Int'l Journal of Open Information Technologies, 2014,2(9):24–27.

[2] Newman S. Building Microservices. Sebastopol: O'Reilly Media, Inc., 2015. 280.

[3] Erl T. Service-Oriented architecture: Concepts, technology and design. http://www.soabooks.com.

[4] Chandola V, Banerjee A, Kumar V. Anomaly detection: A survey. ACM Computing Surveys, 2009,41(3):75–79. [doi: 10.1145/1541880.1541882]

[5] Sigelman BH, Barroso LA, Burrows M, Stephenson P, Plakal M, Beaver D, Jaspan S, Shanbhag C. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, 2010.

[6] Chen MY, Kiciman E, Fratkin E, Fox A, Brewer E. Pinpoint: Problem determination in large, dynamic internet services. In: Proc. of the 2002 Int'l Conf. on Dependable Systems and Networks (DSN 2002). IEEE, 2002. 595–604. [doi: 10.1109/DSN.2002.1029005]

[7] Sambasivan RR, Zheng AX, De Rosa M, Krevat E, Whitman S, Stroucken M, Wang W, Xu L, Ganger GR. Diagnosing performance changes by comparing request flows. In: Proc. of the NSDI. 2011.

[8] Fu Q, Lou JG, Wang Y, Li J. Execution anomaly detection in distributed systems through unstructured log analysis. In: Proc. of the ICDM. 2009. 149–158. [doi: 10.1109/ICDM.2009.60]

[9] ASM. 2016. http://asm.ow2.org/

[10] Instrumentation (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html

[11] Kopetz H, Ochsenreiter W. Clock synchronization in distributed real-time systems. IEEE Trans. on Computers, 1987,C-36(8):933–940. [doi: 10.1109/TC.1987.5009516]

[12] Yao XJ, Gong DW, Li B. Evolutional test data generation for path coverage by integrating neural network. Ruan Jian Xue Bao/Journal of Software, 2016,27(4):828–838 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/004973.htm [doi: 10.13328/j.cnki.jos.004973]

[13] Qian ZS, Miao HK. Specification-Based logic coverage testing criteria. Ruan Jian Xue Bao/Journal of Software, 2010,21(7):1536–1549 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/03615.htm [doi: 10.3724/SP.J.1001.2010.03615]

[14] Bille P. A survey on tree edit distance and related problems. Theoretical Computer Science, 2005,337(1):217–239. [doi: 10.1016/j.tcs.2004.12.030]

[15] King JR, Jackson DA. Variable selection in large environmental data sets using principal components analysis. Environmetrics, 1999,10(1):67–77. [doi: 10.1002/(SICI)1099-095X(199901/02)10:1<67::AID-ENV336>3.0.CO;2-0]

[16] Jolliffe I. Principal Component Analysis. Wiley Online Library, 2002.

[17] TPC-W. http://www.tpc.org/tpcw/default.asp

[18] Zhang W, Wang S, Wang W, Zhong H. Bench4Q: A QoS-oriented e-commerce benchmark. In: Proc. of the 35th Annual Computer Software and Applications Conf. IEEE, 2011. 38–47. [doi: 10.1109/COMPSAC.2011.14]

[19] About the TPC. http://www.tpc.org/information/about/abouttpc.asp

[20] Casale G, Mi N, Smirni E. Model-Driven system capacity planning under workload burstiness. IEEE Trans. on Computers, 2010,59(1):66–80. [doi: 10.1109/TC.2009.135]

[21] Ghanbari S, Amza C. Semantic-Driven model composition for accurate anomaly diagnosis. In: Proc. of the Int'l Conf. on Autonomic Computing (ICAC 2008). 2008. 35–44. [doi: 10.1109/ICAC.2008.33]

[22] Wang T, Wei J, Zhang W, Zhong H, Huang T. Workload-Aware anomaly detection for Web applications. Journal of System Software, 2014,89:19–32. [doi: 10.1016/j.jss.2013.03.060]

[23] Wang T, Wei J, Qin F, Zhang W, Zhong H, Huang T. Detecting performance anomaly with correlation analysis for Internetware. Science China Information Sciences, 2013,56(8):1−15. [doi: 10.1007/s11432-013-4906-6]

[24] Namiot D, Sneps-Sneppe M. On micro-services architecture. Int'l Journal of Open Information Technologies, 2014,2(9):4−8.

[25] Balalaie A, Heydarnoori A, Jamshidi P. Microservices architecture enables DevOps: Migration to a cloud-native architecture. IEEE Software, 2016,33(3):42−52. [doi: 10.1109/MS.2016.64]

[26] Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L. Microservices: Yesterday, today, and tomorrow. arXiv preprint arXiv:160604036, 2016.

[27] Microservices resource guide. http://martinfowler.com/microservices/

[28] Kang H, Chen H, Jiang G. PeerWatch: A fault detection and diagnosis tool for virtualized consolidation systems. In: Proc. of the 7th Int'l Conf. on Autonomic Computing. Washington: ACM Press, 2010. 119−128. [doi: 10.1145/1809049.1809070]

[29] Jiang G, Chen H, Yoshihira K, Saxena A. Ranking the importance of alerts for problem determination in large computer systems. Cluster Computing, 2011,14(3):213−227. [doi: 10.1007/s10586-010-0120-0]

[30] Pham C, Wang L, Tak B, Baset S, Tang C, Kalbarczyk Z, Iyer R. Failure diagnosis for distributed systems using targeted fault injection. IEEE Trans. on Parallel and Distributed Systems, 2017,28(2):503−516. [doi: 10.1109/TPDS.2016.2575829]

[31] Wang T, Zhang W, Ye C, Wei J, Zhong H, Huang T. FD4C: Automatic fault diagnosis framework for Web applications in cloud computing. IEEE Trans. on Systems, Man, and Cybernetics: Systems, 2016,46(1):61−75. [doi: 10.1109/TSMC.2015.2430834]

[32] Chandola V, Banerjee A, Kumar V. Anomaly detection: A survey. ACM Computing Surveys (CSUR), 2009,41(3):15. [doi: 10.1145/1541880.1541882]

[33] Pertet S, Narasimhan P. Causes of Failure in Web Applications. Parallel Data Laboratory, Carnegie Mellon University, 2005. 48−54.

[34] Kiciman E, Fox A. Detecting application-level failures in component-based internet services. IEEE Trans. on Neural Networks, 2005,16(5):1027−1041. [doi: 10.1109/TNN.2005.853411]

[35] Xu W, Huang L, Fox A, Patterson D, Jordan MI. Detecting large-scale system problems by mining console logs. In: Proc. of the ACM SIGOPS the 22nd Symp. on Operating Systems Principles. ACM Press, 2009. 117−132. [doi: 10.1145/1629575.1629587]

[36] Zou H, Hastie T, Tibshirani R. Sparse principal component analysis. Journal of Computational & Graphical Statistics, 2012, 2007(Special):1−30. [doi: 10.1198/106186006X113430]

[37] Kubernetes—Production-Grade Container Orchestration. http://kubernetes.io/

[38] Netflix Open Source Software Center. https://netflix.github.io/

[39] Twitter's finagle library. https://twitter.github.io/finagle/

[40] Rajagopalan S, Jamjoom H. App-Bisect: Autonomous healing for microservice-based apps. In: Proc. of the Usenix Conf. on Hot Topics in Cloud Computing. 2015.

[41] Heorhiadi V, Rajagopalan S, Jamjoom H, Reiter MK, Sekar V. Gremlin: Systematic resilience testing of microservices. In: Proc. of the 36th Int'l Conf. on Distributed Computing Systems (ICDCS). IEEE, 2016. 57−66. [doi: 10.1109/ICDCS.2016.11]

[42] Aguilera MK, Mogul JC, Wiener JL, Reynolds P, Muthitacharoen A. Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review, 2003,37(5):74−89. [doi: 10.1145/1165389.945454]

[43] Reynolds P, Wiener JL, Mogul JC, Aguilera MK, Vahdat A. WAP5: Black-Box performance debugging for wide-area systems. In: Proc. of the 15th Int'l Conf. on World Wide Web. ACM Press, 2006. 347−356. [doi: 10.1145/1135777.1135830]

[44] Bahl P, Chandra R, Greenberg A, Kandula S, Maltz DA, Zhang M. Towards highly reliable enterprise network services via inference of multi-level dependencies. In: Proc. of the ACM SIGCOMM Computer Communication Review. ACM Press, 2007. 13−24. [doi: 10.1145/1282380.1282383]

[45] Reynolds P, Killian CE, Wiener JL, Mogul JC, Shah MA, Vahdat A. Pip: Detecting the unexpected in distributed systems. In: Proc. of the NSDI. 2006. 115−128.

[46] Gschwind T, Eshghi K, Garg PK, Wurster K. Webmon: A performance profiler for Web transactions. In: Proc. of the 4th IEEE Int'l Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems (WECWIS 2002). IEEE, 2002. 171−176. [doi: 10.1109/WECWIS.2002.1021256]

[47] Fonseca R, Porter G, Katz RH, Shenker S, Stoica I. X-trace: A pervasive network tracing framework. In: Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation. USENIX Association, 2007. 20.

[48] Barham P, Isaacs R, Mortier R, Narayanan D. Magpie: Online modelling and performance-aware systems. In: Proc. of the HotOS. 2003. 85−90.

[49] Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. In: Proc. of the OSDI. 2004. 18−27.

[50] Thereska E, Salmon B, Strunk J, Wachs M, Abd-El-Malek M, Lopez J, Ganger GR. Stardust: Tracking activity in a distributed storage system. In: Proc. of the ACM SIGMETRICS Performance Evaluation Review. ACM Press, 2006. 3−14. [doi: 10.1145/1140277.1140280]

[51] Chow M, Meisner D, Flinn J, Peek D, Wenisch TF. The mystery machine: End-to-End performance analysis of large-scale Internet services. In: Proc. of the 11th USENIX Symp. on Operating Systems Design and Implementation (OSDI 2014). 2014. 217−231.

[52] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: Proc. of the 25th Symp. on Operating Systems Principles. ACM Press, 2015. 378−393. [doi: 10.1145/2815400.2815415]

[53] Ghanbari S, Hashemi AB, Amza C. Stage-Aware anomaly detection through tracking log points. In: Proc. of the 15th Int'l Middleware Conf. ACM Press, 2014. 253−264. [doi: 10.1145/2663165.2663319]

[54] Traeger A, Deras I, Zadok E. DARC: Dynamic analysis of root causes of latency distributions. In: Proc. of the ACM SIGMETRICS Performance Evaluation Review. ACM Press, 2008. 277−288. [doi: 10.1145/1375457.1375489]

附中文参考文献:

[12] 姚香娟,巩敦卫,李彬.融入神经网络的路径覆盖测试数据进化生成.软件学报,2008,19(7):1565−1580. http://www.jos.org.cn/1000-9825/004973.htm [doi: 10.13328/j.cnki.jos.004973]

[13] 钱忠胜,缪淮扣.基于规格说明的若干逻辑覆盖测试准则.软件学报,2010,21(7):1536−1549. http://www.jos.org.cn/1000-9825/03615.htm [doi: 10.3724/SP.J.1001.2010.03615]

王子勇(1989－),男,山东临沂人,硕士生,主要研究领域为分布式软件与理论,软件故障检测.

王焘(1982－),男,博士,助理研究员,CCF 专业会员,主要研究领域为软件运行时监控,软件故障检测,自主计算,云计算.

张文博(1976－),男,博士,研究员,博士生导师, CCF 专业会员,主要研究领域为分布式计算,云计算.

陈宁江(1975－),男,博士,副教授,CCF 高级会员,主要研究领域为网络分布计算,软件工程,云计算.

左春(1959－),男,研究员,主要研究领域为分布式软件与理论,云计算,软件工程.