计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(5):1−11 [doi: 10.15888/j.cnki.csa.007888]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于相似度匹配的微服务故障诊断方法①

陈　皓[1], 许源佳[1,2], 王　焘[1,3], 张文博[1,3]

[1](中国科学院 软件研究所, 北京 100190)
[2](中国科学院大学, 北京 100049)
[3](中国科学院 软件研究所 计算机科学国家重点实验室, 100190)
通讯作者: 王　焘, E-mail: wangtao@otcaix.iscas.ac.cn

摘　要: 随着互联网服务的快速发展, 分布式的微服务应用逐渐取代传统的单体应用成为互联网应用的主要形式之一. 微服务应用在具有可伸缩性、容错性、高可用性等优点的同时, 也存在着构建繁琐、部署复杂和维护困难等挑战. 面向云计算环境的微服务监测与运维是当前的研究热点, 但仍然存在粒度较粗、故障定位不准确等缺点. 针对以上问题, 本文提出了一种基于模式匹配的微服务故障诊断方法. 首先, 使用注入代理转发请求流量的方式收集并建模微服务的追踪信息; 然后, 收集系统正常运行下的状态信息, 并通过注入已知故障来收集并刻画故障发生后应用的运行状态; 最后, 将未知故障的执行追踪信息与已知故障的执行追踪信息相匹配, 采用字符串编辑距离衡量相似度以诊断可能的故障原因. 实验结果表明, 该方法可以有效刻画请求的处理执行追踪信息, 以微服务为粒度准确定位应用的故障原因.

关键词: 云计算; 故障诊断; 执行轨迹; 微服务

引用格式: 陈皓,许源佳,王焘,张文博.基于相似度匹配的微服务故障诊断方法.计算机系统应用,2021,30(5):1–11. http://www.c-s-a.org.cn/1003-3254/7888.html

Fault Diagnosis Method Based on Trace Similarity Matching

CHEN Hao[1], XU Yuan-Jia[1,2], WANG Tao[1,3], ZHANG Wen-Bo[1,3]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100049, China)
[3](State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

Abstract: Along with the rapid development of internet services, the distributed microservice-based application has gradually replaced the traditional application as one of the main forms of Internet applications. Distributed microservice-based applications boast scalability, high fault tolerance, and great availability, but they are often challenged by cumbersome installation, complicated deployment, and difficult maintenance. Kubernetes, as the most popular container-based cluster management system, is affected by coarse grains, inaccurate fault location, and other weaknesses. To address the above issues, this study proposes a fault detection method based on trace similarity matching: First, use injecting proxy to forward request traffic to collect tracking information about microservices. Then, collect the state information during normal operation of the system and record the performance of the system after the failure occurs by injecting known faults. Finally, take string edit distance as the standard for the execution tracking models of unknown and known faults. The edit distance serves as a standard to measure the similarity, and the possible cause of failure is

① 基金项目: 国家重点研发计划 (2017YFB1400804); 国家自然科学基金 (61872344); 北京市自然科学基金 (4182070); 中国科学院青年创新促进会人才专项 (2018144)
Foundation item: National Key Research and Development Program of China (2017YFB1400804); National Natural Science Foundation of China (61872344); Natural Science Foundation of Beijing Municipality (4182070); Special Project of Youth Innovation Promotion Association of Chinese Academy of Sciences (2018144)
收稿时间: 2020-08-31; 修改时间: 2020-09-23; 采用时间: 2020-10-09; csa 在线出版时间: 2021-04-28

identified. Experimental results show that the method can accurately describe the processing and execution tracking information of the request and find the cause of system failure with microservices as the granularity.

Key words: cloud computing; fault diagnosis; execution traces; microservices

1 引言

面对互联网时代的海量请求, 短时间内的服务失效会导致用户体验和评价的下降, 而长时间的服务失效甚至会使企业面临严重的经济损失. 随着云计算数据量的迅速增长, 集群规模的不断膨胀, 如何对集群内海量的微服务进行高效、准确的故障检测, 保障集群的高可靠性成为了越来越重要的技术. 服务的监测与故障技术帮助运维人员监测分布式服务的运行情况, 进行资源的调配, 保证整个服务系统的可靠运行.

微服务故障诊断方法主要包括分析度量信息、日志文件和执行追踪等 3 种方法. 首先, 基于度量信息分析的方法收集某个逻辑计量单元或某时间段内的计量值, 可以通过设定固定的指标或是通过一系列运算设定动态变化的指标, 以此作为系统异常的报警规则, 向运维人员发送异常警告, 或是作为集群任务的调度规则[1–3]. 这类方法对于系统内部服务的结构和关系不需要进行了解, 但需要事先知道异常类型和异常特征, 灵活性较差. 其次, 基于日志文件分析的方法收集离散日志文件中的元数据信息, 日志文件记录了系统运行中海量零散事件或请求信息, 通过设定检索模式, 可以找到运行中的错误报告, 有效排查系统的异常原因[4–7]. 这类方法需要收集大量零散的日志文件, 并从中提取出关键的故障信息, 日志收集和信息提取存在滞后性, 难以实时分析系统存在的故障. 最后, 基于执行追踪的方法收集单次请求内的全部信息, 构建系统内部的结构特征, 当系统异常发生时会引起请求处理轨迹发生偏移, 通过对处理轨迹分析以达到异常定位和故障原因诊断的目的[8–10]. 这类方法可用于排查系统性能问题, 但是监测粒度过细会带来巨大的监测和分析资源开销, 存在监测粒度与监测开销之间难以平衡的问题.

综上所述, 现有微服务故障诊断方法存在以下问题: (1) 现有微服务故障诊断方法存在多种监控方式, 其中基于度量信息分析的方法和基于日志文件分析的方法需要预先知道异常的特征信息, 难以应对突发异常, 而基于执行追踪的方法缺少对系统运行指标 (如 CPU、内存、磁盘、网络等) 的监测. (2) 相同请求的执行追踪具有相似性, 缺少衡量执行追踪间的相似度指标, 因此现有技术难以通过分析执行追踪间的相似程度来发现服务异常, 从而造成不能够快速发现和诊断故障的结果. (3) 故障原因的诊断依赖系统历史故障的特征信息, 缺乏对未知故障的可能原因的诊断方法, 因此现有技术需要不断迭代故障特征诊断规则, 对未知故障的诊断依赖运维人员的技术和经验, 从而造成可靠性保障方法灵活性差, 难以应对罕见的系统故障.

针对以上问题, 本文提供一种基于故障相似度的微服务故障诊断方法, 采用预先向系统注入故障的方式, 收集故障的执行追踪, 并通过采用字符串编辑距离表示执行追踪间相似度, 比较已知故障与异常请求间的相似度, 报告可能的故障原因. 相较于现有方法, 本文提出的方法存在以下优点: (1) 基于系统历史运行追踪数据制定故障发生的判断条件, 对于复杂多变的微服务系统, 能够更为灵活准确的判断系统状态; (2) 基于对系统注入故障的表现的真实记录, 因此当生产环境中, 相似的故障再次发生时, 能够通过对系统行为与注入故障的表现进行评估, 快速准确的判断故障原因; (3) 对于系统中罕见故障发生的情况, 本文所提出的方法会将其与已知故障进行比对, 通过对于已知故障原因的分析, 提出可能的相似故障, 运维人员可以此为辅助快速找到系统中真正的故障. 同时, 通过与相关微服务监测方法的结合, 可以减少排除故障所需的时间, 减少实际损失.

2 故障诊断方法

本文提出一种基于故障相似度的微服务故障诊断方法 (图 1), 采用预先向系统注入故障的方式, 收集故障的执行轨迹, 并通过采用字符串编辑距离表示执行轨迹间相似度, 比较已知故障与异常请求间的相似度, 报告可能的故障原因, 具体包括以下 3 个环节:

(1) 执行轨迹建模: 对于分布式系统内服务之间的调用, 采用注入代理的方式拦截和转发流量, 借由分布式追踪系统收集请求信息. 通过收集请求的执行 ID, 父

请求 ID, 执行时间, 调用服务等信息, 可以获得原始的请求信息. 通过对追踪信息进行规范化处理和建立调用关系模型, 可以将请求刻画为相应的有向带权图的形式进行存储. 同时, 对于每一个请求的有向带权图, 都可以通过对于顶点进行映射, 从而转化为一个标识该请求的调用关系的调用字符串.

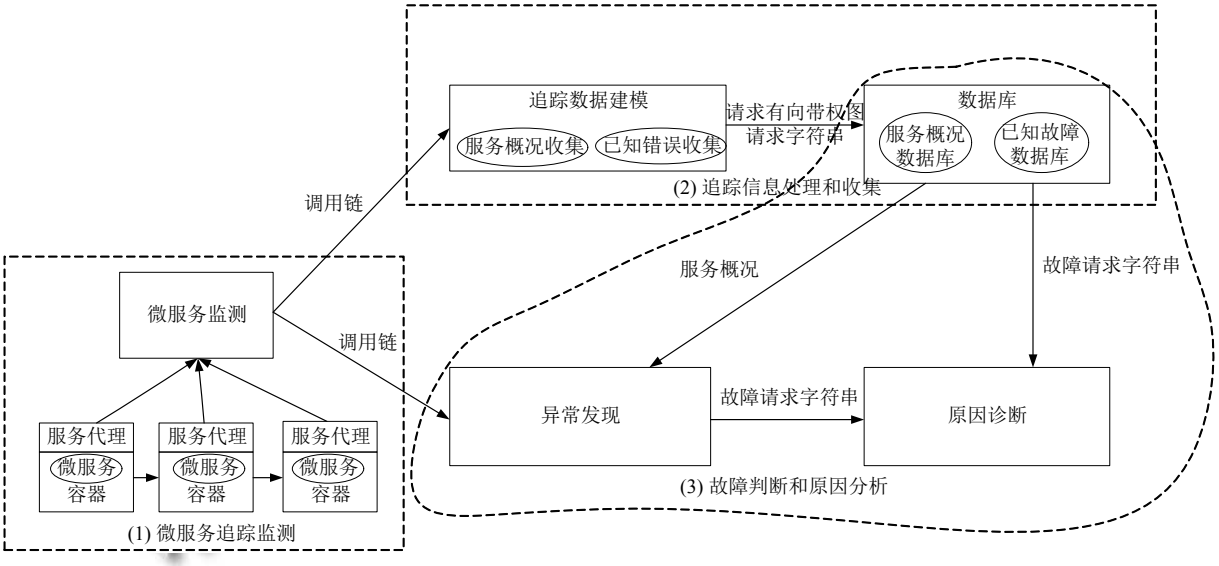


图 1 故障诊断方法总体设计

(2) 系统运行与故障概况统计: 对于被检测系统, 对系统服务正常运行时应用的追踪信息进行收集, 构建正常运行下请求的有向带权图信息进行存储, 作为系统正常运行的参考依据, 设定正常服务间调用请求的正常执行时间区间, 以服务间请求是否超出执行时间区间作为判断请求是否出现异常的标准. 此后, 通过注入已知原因的故障引发系统出现异常, 观察系统应用间执行轨迹的变化关系, 将出现故障后系统内执行轨迹的有向带权图进行收集, 并且对于请求内服务间的调用关系, 通过映射关系转化为请求字符串, 作为系统故障原因的参考依据进行存储.

(3) 未知原因故障发现与原因诊断: 对于运行时的系统, 通过对系统服务间的执行轨迹的不断监测. 一旦系统内出现执行时间异常的请求, 将请求追踪信息作为分析该异常请求的数据来源. 通过对于未知原因故障的请求的有向带权图模型的建立和将请求内服务调用关系转化为调用请求字符串的方式, 通过计算与已知故障的请求字符串之间的编辑距离来衡量未知故障请求与已知故障请求间的相似度, 从而找到能够产生最为相似的调用请求的已知故障, 作为本文判断异常原因结果.

3 关键技术

3.1 执行轨迹建模

对于一条执行轨迹 T, 本文采用有向带权图的结构来表示发送和接收事件的调用关系, 其中: 每一个顶点 S 都对应应用依赖关系中的某一个服务; 对于应用中存在服务 A 和服务 B, 并且服务 A 对服务 B 产生了调用请求, 那么在有向带权图中, 服务 A 和服务 B 对应的顶点 a 和顶点 b 之间建立有向边$\overrightarrow{ab}$, 其权值为调用请求被接受所消耗的时间 t; 根据请求的内容和应用的功能, 对同一服务的多次调用可能会产生不同的调用关系, 因此请求的执行轨迹 T 和对应的带权有向图 D 可能不唯一.

为了记录服务间的依赖关系, 对执行轨迹 T 的每一条追踪信息, 都将表示为:

$$Mi=(MID,requestID,callerID,duration,info) \quad (1)$$

其中, MID 是该服务 id; $requestID$ 为服务请求的 id; $callerID$ 为调用服务的 id; $duration$ 为请求消耗时间; $info$ 包含方法的其他信息, 多元组形式如式 (2):

$$info=(operationName,startTime,tags) \quad (2)$$

其中, $operationName$ 为操作名称; $startTime$ 为开始时间戳; $tags$ 为请求所包含的标签. 通过对依赖关系的还

原, 可以将执行轨迹 T 转化为相应的有向带权图 D=$\{V, E\}$, 其中 V 为顶点 S 的数组, 每一个顶点 S 表示一个服务; E 为邻接矩阵, 被调用的服务间建立权值为消耗时间 t 的有向边. 此外, 对于一条执行轨迹 T, 其中的每一条追踪信息都可以按照调用的顺序, 使用哈希函数将操作名称转化为表示该操作的一个定长字符串 C_m, 根据调用顺序将这些字符串依次拼接, 可得到表示执行轨迹 T 的请求字符串 C.

3.2 追踪信息收集

3.2.1 服务概况收集

本文首先统计系统请求的正确执行, 收集相应的执行轨迹 T_s, 转化为相应的请求有向带权图 D_s, 构建服务概况数据库 (Service Profile DataBase, SPDB), 依次对于系统的请求的正确执行的执行轨迹进行存储. SPDB 的建立不仅是对于系统的系统正常请求的一种记录, 更是能作为系统异常的判定标准. 微服务故障主要分为性能衰减与服务失效两大类, 其中, 性能衰减类的故障是由于资源使用率增高, 使得服务处理效率达到瓶颈, 微服务处理请求所消耗的时间增长, 并且当资源使用率不断增高, 会导致部分微服务处理超时返回错误代码或者微服务间请求处理顺序和调用关系出现异常改变, 其执行追踪表现为微服务执行时间的大幅增长和部分调用关系的异常改变; 服务失效类的故障主要是由于服务未能正确响应请求或服务未被正确配置所引起, 会导致部分请求返回错误信息或者长时间未能响应, 其执行追踪表现为微服务执行时间大幅缩短或大幅增长, 并且存在服务请求调用流程的提前终止或重复请求. 因此, 系统异常往往伴随着执行轨迹中服务执行和调用时间的变化, 本文将采用执行时间作为主要判断的指标.

通过模拟 10 000 次相对稳定的运行环境下被正确处理的请求, 本文收集了 10 000 次相对稳定的运行环境下被正确处理的请求中的追踪信息, 在剔除异常值和缺失值之后, 对收集的 50 037 个追踪中的微服务执行时间进行统计和分析, 相应执行时间如图 2 所示. 事实上, 由于微服务的运行环境并非是一成不变的, 并且会对微服务的运行产生难以预料的影响. 对于相同环境下相同服务的多次相同类型的请求进行统计, 由于网络延时等因素造成的随机性, 仍然会使得调用的时间呈现一定的随机分布, 执行时间近似落在某一个区间之内, 而统计单一因素在请求仍能被正确处理的情况下的波动对执行时间所造成的影响并非本文研究的方向. 然而, 我们将执行时间的分布区间获取的样本中位数为中心, 分成 100 个相同大小的执行时间区间, 并对处在区间内的追踪样本数量进行统计. 我们发现, 对落在每一区间段上的执行时间数量使用 Kolmogorov-Smirnov 检验 (K-S 检验)[11] 进行正态性检验, 相应的 p 值分别为 0.493、0.135、0.259、0.094, 均大于 0.05, 可以认为执行时间的分布区间符合正态分布. 因此, 对于大多数被正确处理的请求, 我们均可以计算得到一个执行时间区间, 将这样的区间端点作为边界值, 落在区间之外的执行轨迹就可以被认为是一种异常. 对于一次新的执行请求, 本文采用式 (1) 和式 (2) 来计算其边界值:

$$t_{\max} = \bar{t} + z_\alpha \times \frac{\sigma}{\sqrt{n}} \tag{3}$$

$$t_{\min} = \bar{t} - z_\alpha \times \frac{\sigma}{\sqrt{n}} \tag{4}$$

其中, $t_{\max}$ 表示正常执行时间的上界, $t_{\min}$ 表示正常执行时间的下界, $\bar{t}$ 表示服务请求的平均执行时间, σ 表示服务请求执行时间的标准差, n 表示样本执行轨迹的数量, z_α 表示正态分布的 α 分位点, 本文采用 95% 的置信概率, 根据文献 [12] 取为 1.96.

当请求的执行时间超出这一区间时, 将被认为发生了一次异常事件, 其执行轨迹的请求字符串 C 将被保留.

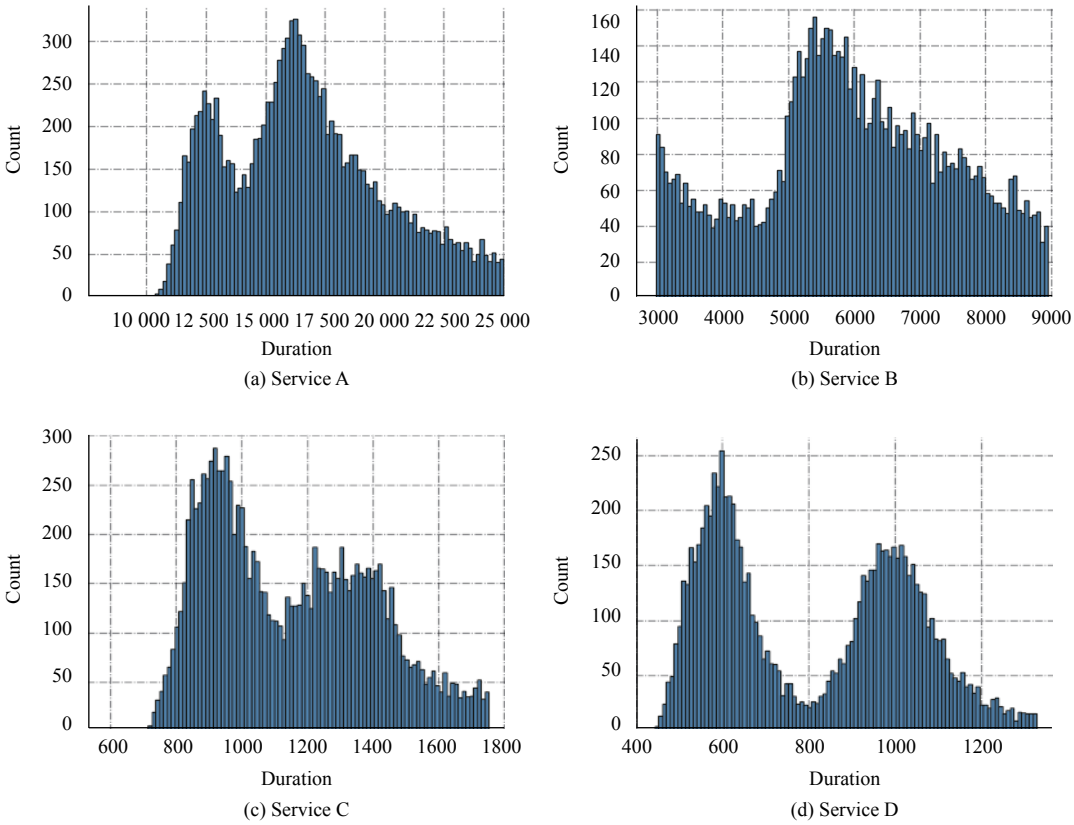
3.2.2 已知错误收集

为了对系统出现故障后的表现进行记录, 本文采用定向注入故障的方式, 预先收集系统失效时的执行轨迹 T_k, 将其转化为相应的请求字符串 C_k 进行存储, 建立已知故障数据库 (Known Faults DataBase, KFDB). KFDB 中不仅可以存储由注入的单个服务的失效和多个服务调用之间的所产生的异常事件执行轨迹, 还能够收集由于集群物理资源异常所引发的执行轨迹, 并且通过本文的处理, 结果利于量化比较相似度. 在系统实际出现失效时, 将出现的故障执行轨迹与 KFDB 中已知故障所产生的所产生的系统执行轨迹进行比较, 通过本文 3.3.2 节所介绍的算法 2 计算出最为相似的故障, 以帮助运维人员判断生产环境中系统失效的实际位置, 最大程度的避免损失.

3.3 故障诊断

系统故障会使请求的执行轨迹的产生可监测的偏移[13], 包括执行轨迹内服务调用关系的改变和执行

时间的波动两种形式. 对于本文所建立的请求的模型, 即请求的有向带权图, 以上两种执行轨迹偏移的形式分别对应有向带权图顶点的改变和有向边权值的波动. 基于以上原理, 提出了故障发现与故障原因诊断的方法.



(a) Service A (b) Service B (c) Service C (d) Service D

图 2 执行时间分布区间

3.3.1 故障检测

对于每一次请求, 经过 3.1 节所介绍的建模方式, 都可以计算出相应的请求有向带权图. 为了处理有向带权图中复杂的调用信息, 判断系统是否发生故障, 本文提出了以下算法思想:

(1) 对于当前请求所对应的有向带权图 D_u, 检索其开始时间最早的顶点 S_1, 作为起始顶点, 表示请求的第一个服务. 对于顶点 S_1, 取其服务名称 N_1, 使用映射函数转化为定长字符串 C_1;

(2) 读取该顶点请求的消耗时间 dt_1 和所连接的下一个顶点 S_2;

(3) 根据调用关系, 进入下一个顶点 S_2, 取其服务名称 N_2, 使用与步骤 (1) 中相同的映射函数转化为定长字符串 C_2;

(4) 查询 SPDB 中 C_1 与 C_2 间执行请求的正常执行区间, 若 dt_1 处于正常执行区间之内, 则判断 S_1 至 S_2 的调用为正常调用, 取顶点请求的消耗时间 dt_2, 将 C_1 添加至请求字符串 C_r 末端; 反之则为异常调用, 将 C_1 添加至请求字符串 C_r 末端;

(5) 重复步骤 (1)~步骤 (4), 对于 T 内的每一个顶点进行检测, 直到当前请求的每一次调用均被检测, 且所有顶点的服务名称均转化为字符串 C 并添加至请求字符串 C_r 中. 如果存在异常, 则将请求字符串 C_r 作为结果返回, 反之则返回空值.

基于以上算法思想, 本文提出故障发现算法如算法 1.

算法 1. 故障发现算法

输入: D_u 未知请求有向带权图
输出: C_r 请求字符串

```
1. function Judgement(Du)
2.   Cr,C1,C2 ← " "
3.   S1 ← Du.V.root
4.   S2 ← null
5.   dt, dtmin, dtmax ← 0
6.   state ← 0
7.     while S1.next is not null do
```

```
8.      S2 ← S1. next
9.      C1← Hash(S1.name)
10.     C2← Hash(S2.name)
11.     dt ← GetArc(Du.E, S1, S2)
12.     dtmin ← GetMinDuration(C1,C2)
13.     dtmax ← GetMaxDuration(C1,C2)
14.     if dtmin > dt or dtmax < dt then
15.       state ← 1
16.     end if
17.     Cr← Cr + C1
18.     S1← S2
19.   end while
20.   if state == 0 then
21.     Cr← null
22.   else
23.     Cr ← C + C2
24.   end if
25.   return Cr
26. end function
```

算法 1 通过以历史系统正常请求的调用时间区间作为判断依据, 能够有效设立有针对性的检测阈值. 其中输入为未知请求有向带权图 D_u= {V, E} , V 为顶点数组, 包含该请求所涉及的所有服务, E 为邻接矩阵, 包含服务间的调用关系和消耗时间. 算法首先从请求开始的顶点, 获取其服务名称和调用关系 (第 3 行~第 9 行), 之后依次将服务名称转化为字符串 (第 9 行、第 10 行), 查询调用时间是否在正常范围之内 (第 11 行~第 16 行), 然后将服务名称添加至请求字符串 (第 17 行) 并重复这样的检测过程, 直至所有顶点均被检测 (第 7 行~第 19 行). 如果存在调用时间超出正常区间, 则返回值为调用字符串, 反之为空值 (第 20 行~第 25 行).

当算法 1 判断系统出现故障后, 将根据所建立的请求模型分析系统可能出现的故障, 本节中所生成的请求字符串将作为描述故障请求重要依据而输入 3.3.2 节的原因诊断.

3.3.2 原因诊断

故障原因诊断基于对系统注入的已知故障进行相似度的判断, 在 3.2.2 节中我们建立了存有已知故障的数据库 KFDB. 为了提取和比较 KFDB 中执行轨迹与未知原因故障的执行轨迹间的相似程度, 我们将故障执行轨迹转化为相应的调用字符串的形式, 采用字符串编辑距离作为相似程度的衡量指标, 详细算法如算法 2 所示.

算法 2 通过已知故障的执行轨迹与未知故障执行轨迹的对比, 返回多个可能的故障原因. 其中输入为请求字符串 C_r, 输出为相似故障 F. 算法对于 KFDB 中记录的每一个故障 F_k, 依次取出请求字符串 C_k, 计算未知请求字符串 C_r 与已知故障请求字符串 C_k 间的编辑距离 (第 6 行~第 10 行). 由于 KFDB 内每一个故障记录了多个已知故障请求字符串, 因此计算出的匹配值为编辑距离的平均值 (第 9 行~第 12 行), 并且返回值为匹配值最小的 KFDB 中已记录的故障 F(第 13 行~第 18 行). 对于有经验的运维人员, 本文所提供的结果可以帮助快速排查系统故障, 查找真正的故障原因.

算法 2. 原因检测算法

输入: C_r 未知请求字符串
输出: F 相似故障

```
1. function SimilarityMatching(Cr)
2.   Cr,Ck ← “”
3.   Fk, F ← null
4.   dk, d ← 0
5.   i ← 0
6.   for each Fk ∈ KFDB do
7.     for each Ck ∈ Fk do
8.       i++
9.       dk ← dk+GetEditDistance(Ck,Cr)
10.    end for
11.    dk ← dk/i
12.    i ← 0
13.    if dk < d then
14.    F ← Fk
15.    d ← dk
16.    end if
17.  end for
18. return F
19. end function
```

4 实验分析

4.1 实验环境

为了验证本文所提出的故障检测方法, 本文采用测试应用的包含 4 组微服务, 微服务之间存在相互调用的关系, 总计包含 36 个微服务实例, 并且被部署在两台应用服务器上, 用于验证本文所提出方法的可行性. 实验平台基于 Kubernetes 和 Istio 框架, 其中 Kubernetes 提供对底层容器自动部署、扩展和管理的容器编排管理功能[14], 而 Istio 通过注入 sidecar 代理的方式拦截微服务之间的网络通讯, 提供统一的微服务连接、安全保障、管理与监控方式[15], 简化了本文对测试系统的

微服务的监控和管理. 包括流量管理, 实验测试应用基于 Istio 所提供的微服务书店应用 Bookinfo[16] 进行实现, 实验系统的架构图如图 3 所示.

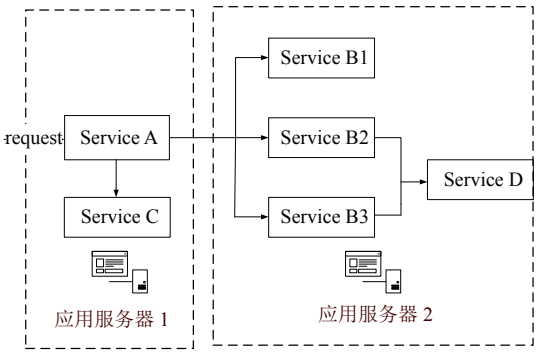


图 3 被测系统架构

当用户请求发送至测试应用后, 首先由 Service A 处理请求, 并且 Service A 会调用 Service C 和随机版本的 Service B 进行处理, 而 v2 和 v3 版本的 Service B 会继续调用 Service D. 测试应用内各微服务功能单一, 可独立运行, 服务之间基于 HTTP/HTTPS 协议的 RESTful API 进行通信协作, 与微服务应用具有的职责单一, 可独立部署、扩展和测试, 通过消息进行交互的特点[17] 相契合. 测试应用运行在 Kubernetes 集群上, 经过服务网格 Istio[18] 对于应用运行和监测的解耦, 其中应用链路数据经过开源链路追踪工具 Jaeger[19] 收集并存储至后端 Elasticsearch[20] 数据库内. 经过本方法处理的请求的有向带权图和相应的调用字符串存储至数据库中. 运行被测系统并发送请求, 收集正常运行时系统的执行轨迹信息, 建立 SPDB. 实验节点的软硬件信息见表 1.

表 1 实验节点配置信息

节点名称	CPU	内存	OS	软件
K8s-node1	QEMU Virtual CPU 4Cores	4 GB 2400 MHz	Ubuntu Server 18.04.4 LTS	Docker 19.03.0 Kubernetes 1.17.3
K8s-node2	QEMU Virtual CPU 4Cores	4 GB 2400 MHz	Ubuntu Server 18.04.4 LTS	Docker 19.03.0 Kubernetes 1.17.3

4.2 执行追踪建模

4.2.1 服务执行信息收集

被测系统在正常请求中会首先调用服务 A, 再由服务 A 对服务 B 和服务 C 进行调用. 其中服务 B 同时有 3 个运行中的版本, B2 和 B3 版本的服务会继续调用服务 D 处理请求. 在实验中, 本文将模拟 1000 次的用户请求, 收集测试系统的正常追踪信息以构建 SPDB. 在经过分布式追踪系统 Jaeger 的收集后, 得到的服务追踪中对 4 个微服务的调用数量和服务之间调用所耗费的平均执行时间如图 4、图 5 所示.

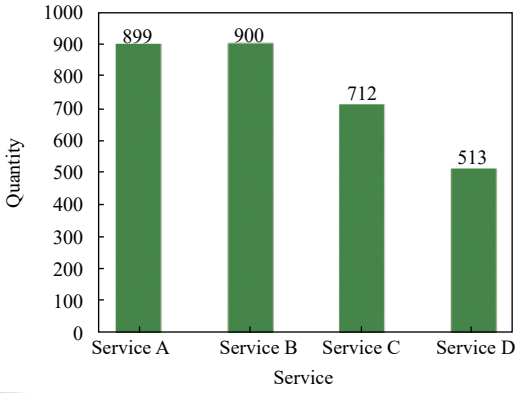


图 4 被测系统收集的微服务追踪数量

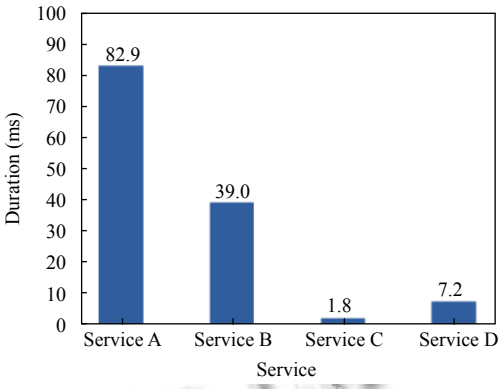


图 5 服务调用的平均执行时间

对于正常请求的执行追踪, 均存在由服务 A 开始的调用其他服务的处理流程. 如图 6 所示, 还原了一次正常请求的有向带权图. 其中, 每一次正确请求的均处在相对稳定的区间内, 并且服务 B 并非每次都需要调用服务 D 处理请求.

4.2.2 故障注入

为了建立 KFDB, 验证方法的有效性, 向被测系统注入了多种故障. 对于每一次的故障注入实验, 均向被测系统发送 500 次请求, 并收集注入故障的系统的追踪信息. 通过对于追踪信息的收集和处理, 对于被测系统的表现进行记录, 建立 KFDB. 注入故障信息见表 2.

其中, 注入节点的节点或服务表示注入故障的应用服务器或服务, 故障描述简述了注入的故障, 有效故障追踪数量表示每次实验所收集到的可处理和分析的追踪数量. 由表 2 可见, 实验中对此被测系统中单次请

求均在 8 次调用内处理完成. 对于生成的系统正常运行时的追踪模型进行与系统设计架构中的调用关系进行人工比对, 可以确认得到请求的追踪模型能够准确的描述每一次的请求信息.

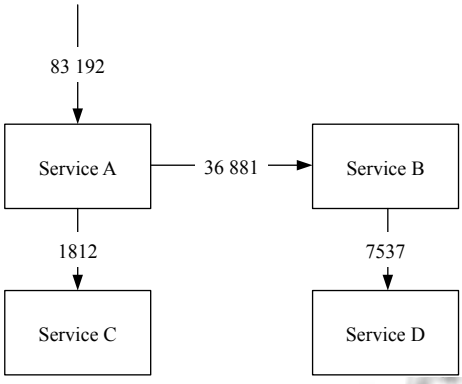


图 6 一次正常请求的有向带权图

表 2 注入故障信息

注入的节点或服务	故障描述	有效故障追踪数量
K8s-node1	CPU高使用率(>80%)	2389
K8s-node2		2239
K8s-node1	内存高使用率(>90%)	2077
K8s-node2		2170
Service A	服务请求延时5 s	1834
Service B1		2156
Service B2		1632
Service B3		1875
Service C		2130
Service D		1577
Service B1、B2、B3	服务pod崩溃重启	1889
Service C		1988
Service D		1726

对于异常请求的执行追踪, 仍然以服务 A 作为调用开始的起点调用其他服务的处理流程. 如图 7 所示, 还原了已知故障 1 中一次请求的有向带权图. 其中, 我们可以观察到, 同样的微服务的执行时间有较明显的变化, 并且由于容器所在节点的不同, 已知故障对于不同的微服务之间的影响幅度并非完全相同. 此外, 由于服务 A 和服务 B 中出现部分超时请求, 后续的服务不能被正确调用, 使得部分执行轨迹出现改变.

4.3 故障诊断

为了检验故障检测方法的有效性, 我们分别在系统处理正常请求的过程中, 引发待测故障, 包含性能衰减与服务失效类型的故障. 对于每一次的故障注入实验, 均向被测系统发送 20 次请求, 并收集注入故障的系统的追踪信息, 待测信息如表 3 所示.

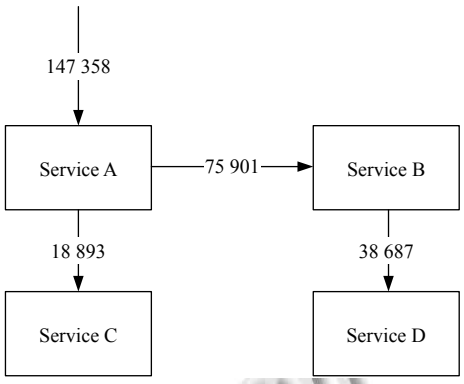


图 7 一次已知故障请求的有向带权图

表 3 待测故障信息

注入节点或服务	故障描述	收集故障追踪信息数量
K8s-node2	CPU高使用率(70%)	106
K8s-node1	内存高使用率(80%)	89
Service A	服务请求延时2 s	78
Service C	服务请求延时7 s	67
Service C	服务pod崩溃重启	83
Service D	服务pod崩溃重启	73

与表 2 类似, 表 3 中注入节点的节点或服务表示注入故障的应用服务器或服务, 故障描述简述了注入的故障, 有效故障追踪数量表示每次实验所收集到的可处理和分析的追踪数量.

对于收集到的待测故障请求的追踪数据, 其中以待测故障 1 为例, 各服务的追踪数据数量与所占比例如图 8、图 9 所示.

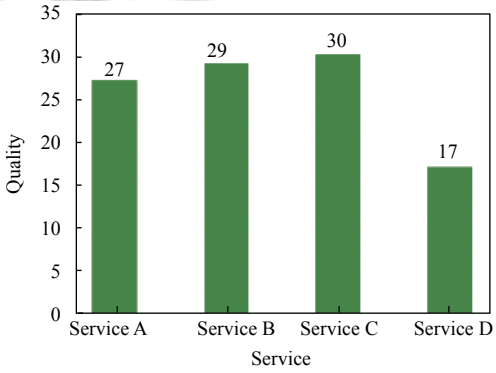


图 8 各服务的追踪数据数量

注入故障会引发服务的性能衰减与失效, 因此由于故障的注入, 系统中应用服务的处理请求所消耗的时间会发生改变. 以待测故障 1 为例, 各服务平均调用时间的变化如图 10 所示.

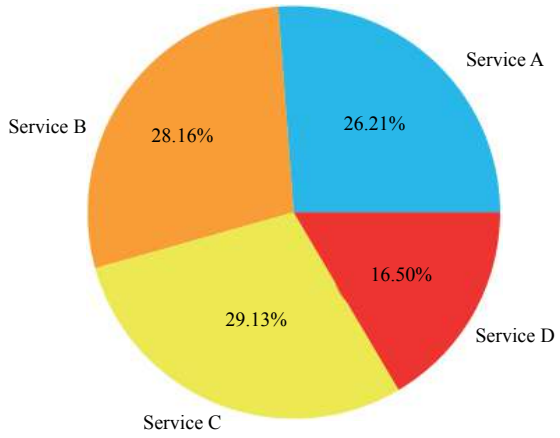


图 9 各服务的追踪数据所占比

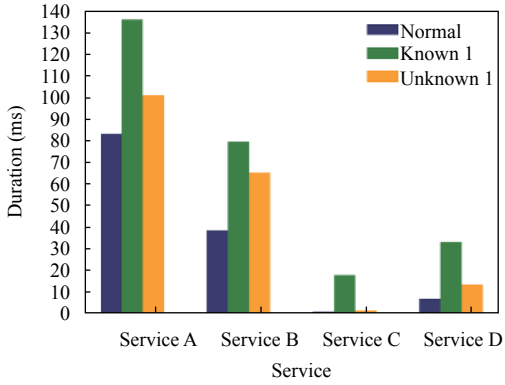


图 10 服务平均执行时间变化

4.4 试验结果分析

4.4.1 结果展示

本文采用查准率 P (precision)、查全率 R (recall) 和 $F1$ 值对本方法的异常发现结果进行评价. 其中:

$$P=\frac{TP}{TP+FP} \tag{5}$$

$$R=\frac{TP}{TP+FN} \tag{6}$$

$$F1=\frac{2\times P\times F}{P+R} \tag{7}$$

其中, 真正例 TP (True Positive) 表示异常发现算法判断为故障时系统确实发生故障的次数, 假正例 FP (False Positive) 表示异常发现算法判断为故障时系统实际未发生故障的次数、假反例 FN (False Negative) 为异常发现算法判断为正常时系统确实发生故障的次数.

对于待检测的故障, 将注入故障的请求的追踪数据和同样次数的系统正常运行下的请求的追踪数据使用本文所提出的异常发现方法进行分析, 其异常发现结果展示如表 4 所示.

表 4 异常发现性能

待测故障编号	P	R	$F1$
1	0.901	1.000	0.952
2	0.864	0.950	0.905
3	0.900	0.900	0.900
4	0.950	0.950	0.950
5	0.714	0.250	0.370
6	0.875	0.350	0.500

同样, 对于原因诊断, 同样采用查准率 P、查全率 R 和 $F1$ 值对结果进行评价. 计算式同式 (3)–式 (5). 其中, 真正例 TP 表示原因诊断算法判断的最可能的已知故障与待测的未知故障的类型相同的次数, 假正例 FP 与假反例 FN 均表示原因诊断算法判断的最可能的已知故障与待测的未知故障的类型不同的次数. 原因诊断结果如表 5 所示.

表 5 原因诊断性能

待测故障编号	P	R	$F1$
1	0.600	0.600	0.600
2	0.550	0.550	0.550
3	0.300	0.300	0.300
4	0.450	0.450	0.450
5	0.650	0.650	0.650
6	0.600	0.600	0.600

4.4.2 结果分析

实验结果表明, 对于服务性能衰减类型的故障, 本文所提出的方法, 本文所提出的故障发现算法可以很好地发现故障, 查准率、查全率、$F1$ 值分别达到 0.886、0.975、0.928, 可见由于性能衰减类的故障会引起服务处理请求时间的相应延长, 因此本文所提出的故障发现算法可以很好地发现这种类型的故障; 而对于服务失效类的异常, 本文所提出的原因诊断算法可以较好地诊断原因, 查准率、查全率、$F1$ 值均为 0.625, 可见由于服务失效类的异常会使得调用请求的处理出现较为明显的变换, 因此本文所提出的原因诊断算法能够较好的发现这种故障的原因.

此外, 本文所提出的方法虽然可以发现故障和检测故障原因, 但是仍有一些情况下会存在对于系统状态和故障原因判断不准确的情况. 经过分析, 未能有效发现的故障主要与 pod 状态异常有关, 主要包括 pod 未能正确提供服务功能的情况, 在实验中的判断准确率. 这是因为本文所提出的异常发现方法基于服务的性能表现, 但是对于可能出现服务失效的应用, 由于请求未能被正确处理而返回, 这一过程与部分应用正常请求所消耗的时间相似, 因此未能正确反映出故障的

发生. 而对于未能正确检测原因的故障, 主要与集群节点物理资源异常所导致的故障有关, 主要包括 CPU 使用率高和内存占用率高等情况. 这是因为资源使用异常类的故障发生时, 仍有部分请求在性能衰减的系统中完成整个处理流程, 返回正确的结果, 因此这一部分的请求虽然处理时间异常, 但仍能描述正确的调用关系, 本文所提出的根据执行追踪所建立模型并未对于这一类请求进行更为详细的描述. 关于这一类故障, 可以结合对于集群节点的物理资源度量和对容器资源的运行时监测进行分析和检测.

5 相关工作

近年来, 微服务软件架构由于其可控制的复杂性、资源可伸缩性、容错性、高可用性等优点逐渐受到各大互联网巨头的青睐. 在服务模型搭建和故障诊断方面, 各有其研究的展开.

在服务模型搭建方面, 文献 [21] 提出了一种评估程序架构信息的方法, 基于模糊的开发代码和不精确的需求, 评估基于所提出框架的微服务开发能否契合原本非功能性需求. 文献 [22] 提出了基于 SDN (Software-Defined Networking) 和 NFV (Network Functions Virtualization) 自我诊断框架, 该框架基于对物理、逻辑、虚拟和服务层的监督内容进行定义, 动态生成诊断模型, 逐渐确定故障区域. 文献 [23] 提出了一种通过分析服务调用链以生成关系依赖图的方法, 通过分析服务依赖关系来找到可能存在风险的调用, 从而找到目标系统存在的异常. 然而, 以上几种方法所建立的模型均没有对异常请求和正常请求分别进行建模和归类, 同时对于系统出现故障的历史表现缺乏记录, 从而当相同的故障发生时诊断效率低下. 本文通过已有的历史模型对未知请求的模型进行比对, 由于使用系统正确处理请求的历史数据模型作为故障判别的标准, 无需修改现有模型即可判断当前系统所处的状态以及可能的产生故障的原因.

在故障诊断方面, 文献 [24] 提出了一种通过记录 Netfiix 的网关 Zuul[25] 活动, 从微服务的请求中收集指标的方法, 这种方法的开销很小, 但是存在着缺乏对于服务之间调用的因果关系计算的过程, 因此会对后续原因诊断造成一定的困扰. 文献 [26] 提出了一种非侵入式的方法, 收集追踪信息和生成日志, 帮助运维人员排除相关故障的方法. 该方法通过捕获网络数据包的形式分析其中的 HTTP 标头, 判断请求的类型并计算跟踪信息. 这种方法由于只依赖网络获取数据包, 不需要对应用进行修改, 但是存在一定的性能消耗, 并且在高负载情况下会减少日志文件的生成. 本文所提出的方法通过预先记录系统正确请求处理的概况和注入故障后系统的行为, 当未知状态请求产生时, 只需将当前请求的追踪信息进行建模比对, 无需对已知请求模型进行修改, 具有较小的资源开销.

6 总结与展望

本文针对微服务监测存在粒度较粗、故障定位不准确等缺点提出一种基于相似度匹配的微服务故障诊断方法. 首先, 使用注入代理转发请求流量的方式收集并建模微服务的追踪信息; 然后, 收集系统正常运行下的状态信息, 并通过注入已知故障来收集并刻画故障发生后应用的运行状态; 最后, 将未知故障的执行追踪信息与已知故障的执行追踪信息相匹配, 采用字符串编辑距离衡量相似度以诊断可能的故障原因. 相较于现有方法, 本文提出的方法存在以下优点: (1) 基于系统历史状态表现分析, 能够灵活准确的判断当前系统状态; (2) 对于历史故障真实还原和记录, 相似故障发生时能够快速匹配和反馈; (3) 对未知故障自动进行相似度匹配, 基于已有故障的表现提出可能的故障原因, 帮助快速排查故障. 同时, 通过与相关微服务监测方法的结合, 可以减少排除故障所需的时间, 减少实际损失.

该方法还存在以下待改进的问题: 首先, 故障注入可能会造成部分罕见故障和复杂故障的遗漏, 后续工作将引入对已检查故障的追踪信息补充录入, 增大故障判断类型的广度; 其次, 检测方法的性能开销与已记录的追踪数量呈正比, 我们后续将对故障注入种类数量和注入点的选取进行研究, 以较小的性能开销获得较高的故障判断准确性.

参考文献

1 Chouliaras S, Sotiriadis S. Real-time anomaly detection of NoSQL systems based on resource usage monitoring. IEEE Transactions on Industrial Informatics, 2020, 16(9): 6042–6049. [doi: 10.1109/TII.2019.2958606]

2 Sotiriadis S, Bessis N, Amza C, *et al*. Elastic load balancing for dynamic virtual machine reconfiguration based on vertical and horizontal scaling. IEEE Transactions on

Services Computing, 2019, 12(2): 319–334. [doi: 10.1109/TSC.2016.2634024]
3 Hochenbaum J, Vallis OS, Kejariwal A. Automatic anomaly detection in the cloud via statistical learning. https://arxiv.org/abs/1704.07706. [2017-04-24].
4 Yuan Y, Shi WC, Liang B, *et al*. An approach to cloud execution failure diagnosis based on exception logs in OpenStack. Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD). Milan, Italy. 2019. 124–131.
5 Xu JM, Chen PF, Yang L, *et al*. LogDC: Problem diagnosis for declartively-deployed cloud applications with log. Proceedings of the IEEE 14th International Conference on E-Business Engineering (ICEBE). Shanghai, China. 2017. 282–287. [doi: 10.1109/ICEBE.2017.52]
6 Jia T, Li Y, Zhang CB, *et al*. Machine deserves better logging: A log enhancement approach for automatic fault diagnosis. Proceedings of 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). Memphis, TN, USA. 2018. 106–111. [doi: 10.1109/ISSREW.2018.00-22]
7 Chuah E, Jhumka A, Alt S, *et al*. Enabling dependability-driven resource use and message log-analysis for cluster system diagnosis. Proceedings of the IEEE 24th International Conference on High Performance Computing (HiPC). Jaipur, India. 2017. 317–327. [doi: 10.1109/HiPC.2017.00044]
8 Zhang SL, Wang Y, Li WJ, *et al*. Service failure diagnosis in service function chain. Proceedings of the 19th Asia-Pacific Network Operations and Management Symposium (APNOMS). Seoul, Republic of Korea. 2017. 70–75. [doi: 10.1109/APNOMS.2017.8094181]
9 Popa NM, Oprescu A. A data-centric approach to distributed tracing. Proceedings of 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). Sydney, Australia. 2019. 209–216. [doi: 10.1109/CloudCom.2019.00039]
10 Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. Proceedings of the 25th Symposium on Operating Systems Principles. New York, NY, USA. 2018. 378–393.
11 Fasano G, Franceschini A. A multidimensional version of the Kolmogorov-Smirnov test. Monthly Notices of the Royal Astronomical Society, 1987, 225(1): 155–170. [doi: 10.1093/mnras/225.1.155]
12 盛骤, 谢式千, 潘承毅. 4 版. 概率论与数理统计. 北京: 高等教育出版社, 2008. 382.
13 Sigelman BH, Barroso LA, Burrows M, *et al*. Dapper, a large-scale distributed systems tracing infrastructure. Google. 2010.http://research.google.com/archive/papers/dapper-2010-1.pdf
14 Kubernetes. Production-grade container orchestration. https://kubernetes.io/. [2019-12-27].
15 What is Istio? https://istio.io/latest/docs/concepts/what-is-istio/. [2020-01-10].
16 Bookinfo application. https://istio.io/latest/docs/examples/bookinfo/. [2020-01-11].
17 Larrucea X, Santamaria I, Colomo-Palacios R, *et al*. Microservices. IEEE Software, 2018, 35(3): 96–100. [doi: 10.1109/MS.2018.2141030]
18 Istio. https://istio.io/. [2020-01-10].
19 Jaeger: Open source, end-to-end distributed tracing. https://www.jaegertracing.io/. [2020-04-06].
20 Meet the core products —All free and open. https://www.elastic.co/elastic-stack. [2020-04-06].
21 Márquez G, Lazo Y, Astudillo H. Evaluating frameworks assemblies in microservices-based systems using imperfect information. Proceedings of 2020 IEEE International Conference on Software Architecture Companion (ICSA-C). Salvador, Brazil. 2020. 250–257. [doi: 10.1109/ICSA-C50368.2020.00049]
22 Sánchez JM, Ben Yahia IG, Crespi N. Self-modeling based diagnosis of services over programmable networks. Proceedings of 2016 IEEE NetSoft Conference and Workshops (NetSoft). Seoul, Republic of Korea. 2016. 277–285. [doi: 10.1109/NETSOFT.2016.7502423]
23 Ma SP, Fan CY, Chuang Y, *et al*. Using service dependency graph to analyze and test microservices. Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC). Tokyo, Japan. 2018. 81–86. [doi: 10.1109/COMPSAC.2018.10207]
24 Pina F, Correia J, Filipe R, *et al*. Nonintrusive monitoring of microservice-based systems. Proceedings of the IEEE 17th International Symposium on Network Computing and Applications (NCA). Cambridge, MA, USA. 2018. 1–8. [doi: 10.1109/NCA.2018.8548311]
25 Zuul. https://github.com/Netflix/zuul. [2020-09-01].
26 Cinque M, Della Corte R, Pecchia A. Microservices monitoring with event logs and black box execution tracing. IEEE Transactions on Services Computing. [doi: 10.1109/TSC.2019.2940009]