软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(1):118−135 [doi: 10.13328/j.cnki.jos.006827]

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

微服务系统服务依赖发现技术综述*

张齐勋[1,2], 吴一凡[2], 杨 勇[1], 贾 统[4], 李 影[2,3], 吴中海[2,3]

[1](北京大学 信息科学技术学院, 北京 100871)

[2](北京大学 软件与微电子学院, 北京 102600)

[3](北京大学 软件工程国家工程研究中心, 北京 100871)

[4](北京大学 人工智能研究院, 北京 100871)

通信作者: 李影, E-mail: li.ying@pku.edu.cn

摘 要: 微服务架构得到了广泛的部署与应用, 提升了软件系统开发的效率, 降低了系统更新与维护的成本, 提高了系统的可扩展性. 但微服务变更频繁、异构融合等特点使得微服务故障频发、其故障传播快且影响大, 同时微服务间复杂的调用依赖关系或逻辑依赖关系又使得其故障难以被及时、准确地定位与诊断, 对微服务架构系统的智能运维提出了挑战. 服务依赖发现技术从系统运行时数据中识别并推断服务之间的调用依赖关系或逻辑依赖关系, 构建服务依赖关系图, 有助于在系统运行时及时、精准地发现与定位故障并诊断根因, 也有利于如资源调度、变更管理等智能运维需求. 首先就微服务系统中服务依赖发现问题进行分析, 其次, 从基于监控数据、系统日志数据、追踪数据等 3 类运行时数据的角度总结分析了服务依赖发现技术的技术现状; 然后, 以基于服务依赖关系图的故障根因定位、资源调度与变更管理等为例, 讨论了服务依赖发现技术应用于智能运维的相关研究. 最后, 对服务依赖发现技术如何准确地发现调用依赖关系和逻辑依赖关系, 如何利用服务依赖关系图进行变更治理进行了探讨并对未来的研究方向进行了展望.

关键词: 服务依赖; 故障诊断; 微服务

中图法分类号: TP311

中文引用格式: 张齐勋, 吴一凡, 杨勇, 贾统, 李影, 吴中海. 微服务系统服务依赖发现技术综述. 软件学报, 2024, 35(1): 118–135. http://www.jos.org.cn/1000-9825/6827.htm

英文引用格式: Zhang QX, Wu YF, Yang Y, Jia T, Li Y, Wu ZH. Survey on Service Dependency Discovery Technologies for Microservice Systems. Ruan Jian Xue Bao/Journal of Software, 2024, 35(1): 118–135 (in Chinese). http://www.jos.org.cn/1000-9825/6827.htm

Survey on Service Dependency Discovery Technologies for Microservice Systems

ZHANG Qi-Xun[1,2], WU Yi-Fan[2], YANG Yong[1], JIA Tong[4], LI Ying[2,3], WU Zhong-Hai[2,3]

[1](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](School of Software and Microelectronics, Peking University, Beijing 102600, China)

[3](National Engineering Research Center for Software Engineering, Peking University, Beijing 100871, China)

[4](Institute for Artificial Intelligence, Peking University, Beijing 100871, China)

Abstract: Microservice architectures have been widely deployed and applied, which can greatly improve the efficiency of software system development, reduce the cost of system update and maintenance, and enhance the extendibility of software systems. However, However, microservices are characterized by frequent changes and heterogeneous fusion, which result in frequent faults, fast fault propagation, and great influence. Meanwhile, complex call dependency or logical dependency between microservices makes it difficult to locate and diagnose faults timely and accurately, which poses a challenge to the intelligent operation and maintenance of microservice architecture

* 基金项目: 广东省重点领域研发计划 (2020B010164003); 华为产学研合作课题 2021–2022
收稿时间: 2021-12-02; 修改时间: 2022-08-22, 2022-10-09; 采用时间: 2022-11-11; jos 在线出版时间: 2023-05-31
CNKI 网络首发时间: 2023-06-02

systems. The service dependency discovery technology identifies and deduces the call dependency or logical dependency between services from data during system running and constructs a service dependency graph, which helps to timely and accurately discover and locate faults and diagnose causes during system running and is conducive to intelligent operation and maintenance requirements such as resource scheduling and change management. This study first analyzes the problem of service dependency discovery in microservice systems and then summarizes the technical status of the service dependency discovery from the perspective of three types of runtime data, such as monitoring data, system log data, and trace data. Then, based on the fault cause location, resource scheduling, and change management of the service dependency graph, the study discusses the application of service dependency discovery technology to intelligent operation and maintenance. Finally, the study discusses how service dependency discovery technology can accurately discover call dependency or logical dependency and use service dependency graph to conduct change management and predicts future research directions.

Key words: service dependency; fault diagnosis; microservice

随着分布式软件系统的规模越来越大, 结构越来越复杂, 其开发与维护都面临着巨大的挑战. 因此, 越来越多的企业采用微服务架构将其承载核心业务的软件系统进行重构[1]. 在微服务架构的软件系统中, 庞大的软件系统按照低耦合的原则被拆分为多个可独立部署的微服务, 不同的微服务可以由不同的团队使用不同的技术栈进行开发与维护, 微服务之间通过 RESTful、RPC (remote procedure call)、HTTP 等方式进行通信, 软件系统的不同功能通过多个不同的微服务之间的共同协作实现. 微服务架构极大地提升了软件系统开发的效率, 降低了系统更新与维护的成本, 提高了系统的可扩展性. 微服务架构的软件系统具有以下特点: (1) 变更频繁. 系统更新成本的降低使得每个微服务都在不断更新以满足系统需求的不断变化, 且不断有新的微服务动态地加入与退出微服务架构软件系统; (2) 依赖复杂. 单个微服务的功能单一使得各个系统功能的实现都会产生大量微服务之间复杂的交互; (3) 异构融合. 不同微服务往往采用不同的技术栈实现, 涉及大量不同编程语言、中间件与运行环境的融合. 频繁的变更导致微服务架构的软件系统中故障频发, 而微服务之间复杂的交互使得单个服务的故障在整个系统中迅速传播, 导致大量的服务失效, 且难以定位故障的根因, 微服务实现的异构性则使得依据专家知识从系统全局对故障进行理解与定位难以实现. 准确地发现微服务架构软件系统中各个微服务之间的复杂的依赖关系, 构建全面的服务依赖关系图, 能够帮助开发与运维人员高效、精准地发现与定位系统中的故障并进行根因分析, 也有利于在系统部署的过程中根据服务依赖优化微服务的部署策略, 提高服务质量, 对提高微服务架构软件系统的运维效率具有重要的研究意义.

服务依赖的发现早期工作主要依靠商业软件如 IBM Tivoli[2]和微软的 MOM (Microsoft operations manager)[3]等, 依赖人工根据专家知识对分布式软件系统中服务间的依赖关系进行配置与管理, 也有相关工作通过挖掘服务的配置文件及软件的安装信息构建服务之间的依赖关系图[4]. 由于面向服务的架构 (service-oriented architecture)及近年来微服务架构的出现, 软件系统中存在大量的服务依赖, 且大量服务依赖并非在开发与部署时静态地指定, 而是在运行时通过动态服务绑定机制 (即服务发现, service discovery) 产生, 系统频繁的变更加剧了服务软件系统中服务依赖的动态性, 使得基于人工的方法以及通过挖掘静态的服务配置文件构建服务依赖关系图的方法不再可行, 只能在软件系统部署运行后基于运行时数据进行自动化的构建. 图 1 给出了自 2000 年以来自动化构建服务依赖关系图方向的学术论文数量、发表源及其发表年限. 图 1 中可见, 近 20 年来, 均有稳定数量的服务依赖发现领域的研究成果发表在计算机领域顶级、重要的会议与期刊中. 如何基于系统运行时数据自动化地构建服务依赖关系图逐渐成为学术界与工业界共同关注的重点问题.

随着微服务架构被更加广泛地应用, 微服务架构软件系统变更频繁、依赖复杂、异构融合的特点对服务依赖发现技术的实时性、准确性和通用性提出了新的要求, 使得微服务系统中服务依赖发现仍然面临着很大的挑战. 目前尚没有对基于运行时数据的服务依赖发现技术及其在微服务系统中应用的综述. 本文从 3 类运行时数据, 即监控数据、系统日志数据、追踪数据的角度综述了服务依赖发现技术的发展现状, 然后介绍了近年来基于服务依赖发现的应用研究工作, 最后对微服务系统中服务依赖发现领域未来值得关注的问题和研究方向进行了探讨.

本文第 1 节对服务依赖发现相关的概念进行阐述. 第 2 节给出服务依赖发现领域已有的研究工作的分类方法并对相关工作进行详细的介绍与分析, 并总结其优缺点. 第 3 节分析了学术界与工业界基于服务依赖发现技术的应用研究. 第 4 节总结了服务依赖发现技术的相关工程应用. 第 5 节对服务依赖发现技术未来的研究方向进行展望.

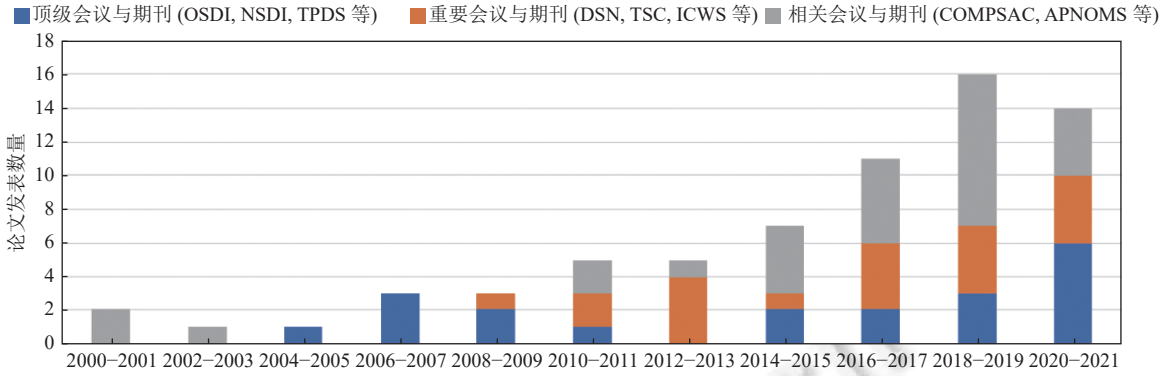


图 1 论文发表数量、出处及发表年份分布

1 问题描述

在微服务架构软件系统中, 微服务之间的依赖关系可用服务依赖关系图进行表示. 服务依赖关系图[5−7]是一个有向图 $G=<V,E,W>$, 图中的每个顶点 V_i 代表一个微服务, 每一条有向边 $E_k=<V_i,V_j>$ 代表两个服务 V_i 与 V_j 之间存在依赖关系且 $V_i \rightarrow V_j$ (即服务 V_i 依赖服务 V_j), W_k 是每条有向边 E_k 的权重, 表示依赖关系成立的置信度或者依赖关系的强弱. 但在不同的研究工作中, 服务依赖发现的具体含义由于历史原因有所不同, 本节对服务依赖发现过程中可能出现的歧义进行梳理.

● 服务. 在微服务架构软件系统中, 服务即指微服务. 但在已有的服务依赖发现相关研究工作中, 并没有一个通用且标准的关于服务的定义, 所以在不同的研究工作中, 服务依赖发现中的“服务”的具体含义可能有所不同, 但基本可以划分为 3 类: 由 IP 和 Port 代表的服务, 组件或者应用, 虚拟机. 在文献 [5−14] 中, 服务或被定义为<IP, Port> 这样的二元组, 或被定义为<IP, Port, Protocol> 的三元组. 在文献 [15−20] 中, 服务即组件/应用, 组件是分布式软件系统中可被独立部署的最小单元. 而将虚拟机作为服务依赖发现的研究对象[21,22]时, 通常是基于假设: 每个虚拟机中仅部署一个服务, 因此虚拟机之间的依赖关系也就代表服务之间的依赖关系. 在一对服务依赖关系中, 服务按照是依赖的一方还是被依赖的一方可以划分为依赖服务 (depending service) 和被依赖服务 (depended service).

● 依赖. 服务依赖发现中的依赖关系有两种, 调用依赖关系 (local-remote dependency) 和逻辑依赖关系 (remote-remote dependency)[8,9]. 调用依赖关系指一个服务 V_i 为完成对该服务的请求的响应, 对其他服务如 V_j 的调用关系, 是微服务系统中最常见的依赖关系. 如图 2 所示在一个典型的开源微服务系统[23]中所发现的部分服务依赖关系中, CheckoutService 为完成结账服务, 会分别调用 CartService、PaymentService 和 ShipmentService 完成下单、支付和邮寄功能, 那么 CheckoutService 依赖于 CartService、PaymentService 以及 ShipmentService, 依赖类型为调用依赖. 逻辑依赖关系是指一个服务 V_i 完成对该服务的请求响应是以另一个服务 V_j 完成对指定请求响应为前提的逻辑先后关系. 如图 2 所示的服务依赖关系中, ShipmentService 为完成邮寄服务, CheckoutService 首先需要调用 PaymentService 完成支付, 那么 ShipmentService 依赖 PaymentService, 依赖类型为逻辑依赖. 依赖关系是可以传递的[8], 即 $(V_i \rightarrow V_j) \wedge (V_j \rightarrow V_k) \Rightarrow V_i \rightarrow V_k$, 根据依赖关系是否是由其他依赖关系的传递而衍生, 又可将依赖关系分为直接依赖 (direct dependency) 关系与间接依赖 (indirect dependency) 关系, 所有间接依赖关系都可以通过直接依赖关系传递获得, 因此为了保持服务依赖图的统一与简洁, 服务依赖图中依赖关系视为直接依赖关系. 除此之外, 服务依赖发现方法通常会基于不同算法赋予依赖关系一个数值来衡量依赖关系的强弱或依赖关系存在的置信度.

● 服务依赖发现. 服务之间的依赖关系会反映在运行时数据中, 使得存在依赖关系的两个服务的运行时数据之间存在一定的规律, 因此服务依赖发现即从运行时数据中识别与推断服务之间的依赖关系, 构建服务依赖关系图的过程. 在不同的运行时数据上使用不同的方法, 可以直接识别出服务之间的依赖关系, 或者通过计算服务之间的相关性推断两个服务的依赖关系. 相关性是依赖关系的一种近似, 服务之间的依赖关系会导致服务之间的运行

数据上存在相关性, 但相关性并不仅由依赖关系导致, 可能是由于两个服务依赖于同一个服务或运行时数据中的噪音等原因产生的, 所以两个服务之间存在相关性并不能直接说明两个服务之间存在依赖关系[8].

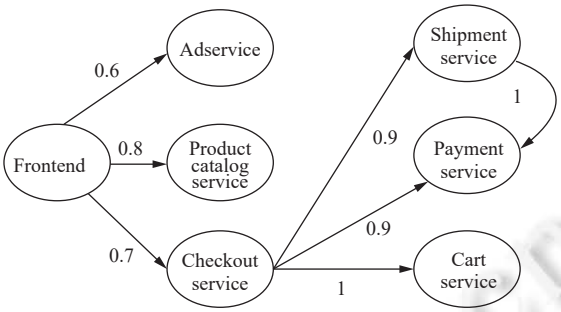


图 2 微服务系统中的服务依赖

2 服务依赖发现

伴随着服务架构从少量、静态、简单向海量、动态、复杂演变, 服务依赖发现方法也从以网络流量为核心的静态服务依赖发现向以多源运行时数据为核心的动态复杂服务依赖进行演变. 伴随着服务网格的出现, 轻量级的网络代理与应用程序之间存在海量动态拓扑关系, 这些异构变化的网络代理与应用程序, 进一步提升了微服务系统的复杂性, 进而导致服务依赖发现方法的演变趋势向海量异构服务节点数据快速分析和实时依赖变化感知的方向发展.

本文从多源运行时数据角度对服务依赖发现方法进行综述分析. 系统运行时数据可以分为 3 类: 监控数据、系统日志数据与追踪数据. 监控数据是由监控工具在系统运行时获取的用以表征系统运行状况的数据, 包括网络通信包 (packet) 数据、资源使用数据如 CPU/内存等的使用量、业务统计指标如请求响应时间与吞吐量等. 系统日志数据是由开发人员在开发时添加的日志打印语句在系统运行时产生的用以记录程序运行状态及相关变量信息的半结构化文本数据. 追踪数据是由分布式追踪技术[24]产生的用以刻画请求在分布式软件系统中端到端的处理过程的数据. 图 3 展示了服务依赖发现的基本流程. 首先, 多数服务依赖发现方法依赖于运行时数据的分布变化相关性, 为加速和加剧分布变化, 需要利用故障或干扰注入工具对微服务系统进行故障和干扰注入. 然后, 收集微服务系统产生的监控、系统日志和追踪数据并利用这些数据发现微服务实例和微服务依赖关系. 最后, 根据服务依赖关系构建服务依赖关系图. 相关研究工作分别基于 3 类不同运行时数据, 提出了不同自动化构建服务依赖关系图的方法.

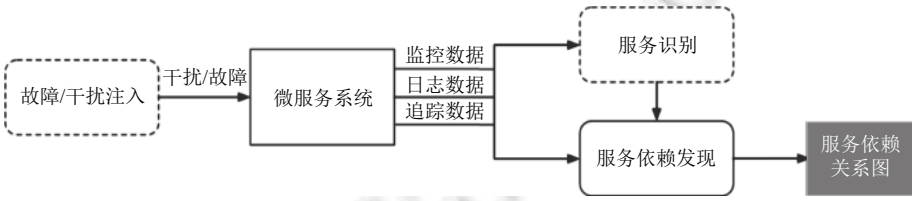


图 3 服务依赖发现流程

2.1 基于监控数据的服务依赖发现

2.1.1 基于网络通信包数据的服务依赖发现

基于网络通信包数据的服务依赖发现方法利用存在依赖关系的两个微服务的网络通信消息中存在特定交互模式与时空上相关性的特点, 通过监听与解析网络传输层网络包数据, 使用统计方法从中推断服务之间的依赖关系.

基于网络通信包数据的服务依赖发现首先利用网络包监控工具获取每个节点上所有 TCP packets 与 UDP packets, 从每个 packet 中提取一个五元组<SrcIP, SrcPort, DestIP, DestPort, Protocol>, 其中 SrcIP, SrcPort, DestIP, DestPort, Protocol 分别表示一个 packet 的源端 IP, 源端端口, 目标 IP, 目标端口与传输层协议; 然后根据五元组将在一定时间窗口内所有拦截到的 packets 划分为不同流 (flow)/通道 (channel)/会话 (session), 同一个流中 SrcIP, SrcPort, DestIP, DestPort 是相同的 (或者源端 IP 和源端端口与目标 IP 和目标端口交换), 进而得到表征每一个流的七元组<SrcIP, SrcPort, DestIP, DestPort, Protocol, startTime, endTime>, TCP 流的 startTime 是建立 TCP 连接 3 次握手时第 1 个 packet 的时间戳, endTime 是关闭 TCP 连接 4 次握手时最后一个 packet 的时间戳, UDP 流的 startTime 是最早出现该五元组 pakcet 的时间戳, endTime 是在大于指定的时间间隔内不再出现该五元组 packet 的最后一个 packet 的时间戳; 构建系统中每个节点的所有流之后, 不同文献采用不同方法计算两个由<IP1, Port1> 和 <IP2, Port2> 代表的两个服务是否存在依赖关系以及依赖关系成立的概率.

大量的研究工作聚焦于从网络通信数据包中发现服务之间的逻辑依赖关系. 文献 [25] 其所提方法中的 4 个步骤可以被视为此类方法的一般过程: 1) 流量获取 (traffic acquisation), 使用 Wincap[26]等工具获取并解析网络包; 2) 流识别 (flow identification), 将所有网络包划分到不同的 TCP 和 UDP 流中; 3) 流排序 (flow sequencing), 将每个节点上监控到的不同流在一定时间窗口内的共现/交织进行统计; 4) 依赖分析 (dependency analysis), 汇总多个节点上获取的不同流所代表的服务之间的共现/交织数据, 使用不同方法计算两个服务之间是否存在依赖关系及依赖关系的强弱. 文献 [6,7] 是最早利用网络包分析构建服务逻辑依赖图的工作, 提出了两种方法: Constellation 和 AND (analysis of network dependencies), 用于从网络包数据中推断服务之间的逻辑依赖关系. 两种方法在每个节点上按照该节点是通道的源还是目标将每个通道划分为输入通道 (input channel) 和输出通道 (output channel), 并根据通道是否已经关闭将通道的状态划分为 active 和 inactive 两种. Constellation 通过使用有监督的机器学习方法如朴素贝叶斯 (naïve Bayes), 在每个节点上学习一定长度时间窗口内的数据, 根据多个输入通道是否 active 来预测输出通道是否 active, 依据预测模型中有效的特征 (即某个输入通道是 active 能较大程度上预测某个输出通道是 active), 表示该输入通道<SrcIP1, SrcPort1>所代表的服务依赖于所要预测的输出通道的<DestIP2, DestPort2>所代表的服务, 其依赖强度由预测模型中的每个特征参数表示. AND 则通过在每个节点上计算一定时间窗口内 (100 ms) 任意两个 Active 的输出通道中的<DestIP1, DestPort1>和<DestIP2, DestPort2>所代表服务共同出现频率来判断两个服务是否存在依赖关系, 并根据<DestIP1, DestPort1>与<DestIP2, DestPort2>共同出现时的先后顺序来判断服务依赖方向, 服务依赖强度可以用所有节点上观察到的两个服务共同出现的概率来表示.

Sherlock[5]提出了推理图 (inference graph) 概念用于表示服务依赖关系图, 是对文献 [6] 中提出的 Lesile Graph 的扩展, 将服务依赖关系图中节点分为 3 类: 根因结点 (root cause node), 观察结点 (oberservation node) 和元结点 (meta node). 其中根因结点表示服务依赖发现所关注的服务, 观察结点表示安装 Agent 观察并解析网络包数据的节点, 元结点表示在系统中用于负载均衡 (load-balancing) 和失效转移 (failover) 的节点. Sherlock 自动发现服务依赖方法与文献 [6] 相似, 利用在每个 Meta Node 上观测到的不同流所表示的<DestIP, DestPort>服务, 在一定时间窗口内 (10 ms) 共现的条件概率表示服务之间是否存在依赖以及依赖的强弱. Sherlock 构建的网络依赖关系中, 表示负载均衡与失效转移节点的元结点与其他根因结点的依赖关系需要人工进行构建.

MacroScope[11]将服务依赖分为静态依赖 (static dependency) 和瞬时依赖 (transient dependency), 分别表示使用固定端口的服务和使用临时端口的服务. MacroScope 仅关注自动构建应用以及服务之间的静态依赖, 且关注多层次的服务依赖关系. 其多层次是指包括服务 (以<Port, Protocol>表示) 和服务实例 (以<IP, Port, Protocol>表示) 的依赖关系. MacroScope 除拦截网络包数据用于构建服务之间的依赖关系外, 同时拦截每个端口的进程绑定信息, 确定<IP, Port>所代表的进程名称, 准确地获取应用级别的服务名称信息, 帮助理解构建的服务依赖关系图在应用层面的含义.

其中一部分相关工作则侧重于使用不同的统计方法来衡量不同的流所代表的服务之间依赖关系是否成立, 以及依赖关系的强弱. 文献 [27] 通过计算两个流所代表的目标端服务的皮尔逊相关系数 (Pearson correlation coefficient) 来衡量两个服务之间相关性, 进而推断两个服务的依赖关系. 文献 [14] 提出使用矩阵分解 (matrix factorization) 方

法计算服务之间依赖关系强度的方法. 文献 [10] 提出了 eXpose 方法用于从网络包数据中发现服务依赖, 其方法同样与文献 [6] 相似, 在每个节点上一定时间窗口 (1 s) 内, 学习两个流所代表的目标端服务的依赖关系, 不同于文献 [5–7], eXpose 不计算两个流所代表的目标端服务共现频率或者条件概率, 而是利用信息论中 JMeasure 方法, 解决使用共现频率或条件概率时, 服务间依赖的方向难以判断以及受噪音影响大的问题, 能够更加准确地判断依赖关系及方向.

不同于发现服务之间的逻辑依赖关系, 也有研究人员关注如何从网络包数据中发现服务之间的调用依赖关系. 文献 [28] 直接根据每个 TCP 流中<SrcIP, SrcPort>和<DestIP, DestPort>挖掘两个服务之间的调用依赖关系. 但直接根据 TCP/UDP 流判断服务之间是否存在调用依赖关系无法判断依赖关系的强弱. NSDMiner[12]是最早提出从网络包数据中推断服务之间调用依赖关系的工作, NSDMiner 基于假设: 在服务 A 与服务 B 的流状态是 active 期间 (即以 A 为源, 以 B 为目标的流), 如果服务 B 产生了新的与服务 C 的流 (以 B 为源, 且 C 为目标), 且服务 B 与服务 C 的流在服务 A 与服务 B 的流关闭之前关闭 (即服务 B 与服务 C 的流嵌套于服务 A 与服务 B 之间的流), 那么服务 B 依赖于服务 C, 且可以据此判断 B 与 C 之间依赖关系的强弱. NSDMiner 直接使用每个节点上在服务 A 与服务 B 的流处于 Active 期间, 服务 B 与服务 C 的流嵌套比例来计算服务 B 与服务 C 之间依赖关系强弱. 文献 [13] 对 NSDMiner 计算两个服务之间调用依赖关系强弱方法进行了改进, 提出了一种基于对数 (logarithm-based) 的服务依赖强弱计算方法. 除此之外, 文献 [13] 还针对访问频率较低的服务, 其服务依赖关系难以构建的问题, 基于假设: 相似度高的服务通常具有相似的服务依赖关系, 提出了一种基于服务依赖相似度推断访问频率较低服务的服务依赖关系的方法, 服务之间相似度根据服务之间共同依赖服务的占比计算. 文献 [23] 同样对存在负载均衡及失效转移情况下, 服务依赖关系图的构建提出了一种基于规则的解决方法, 改进了相关工作[9,12]中这类依赖需要人工进行构建的问题. 文献 [29] 则使用条件概率替代了 NSDMiner 中基于比例的服务依赖关系判断方法, 并对 NSDMiner、Orion[9]和 Sherlock[5]用于发现服务依赖的准确性进行了对比实验. 针对文献 [12,13] 仅利用流之间时空上相关性判断服务之间的依赖关系, 受流的随机交叉影响较大的问题, 文献 [30] 则提出了一种基于传递熵 (transfer entropy) 来判断服务之间调用依赖关系的方法, 其在 NSDMiner 的基础上, 根据两个服务之间的通信字节数计算两个服务之间的传递熵, 当传递熵大于指定阈值时, 两个服务之间存在依赖关系, 且依赖强弱为得到的传递熵.

2.1.2 基于资源使用数据的服务依赖发现

基于资源使用数据的服务依赖发现技术利用存在依赖关系的两个服务之间资源使用在时间序列存在相似性的特点, 通过不同算法计算不同服务在一维或多维的资源使用时间序列数据上的相似度, 推断任意两个服务之间的相似度即服务依赖的强弱.

文献 [21] 获取每个服务 CPU 使用量的时间序列, 使用 auto-regressive (AR) 模型对每个服务 CPU 使用峰值之间的关系进行建模, 然后使用欧氏距离 (Euclidean distance) 衡量不同服务之间的 AR 模型的相似度, 进而使用 K-means 算法根据不同服务之间的相似度将服务进行聚类, 聚到同一簇的任意两个组件之间存在依赖关系, 其依赖关系的强弱即两者之间的相似度. 文献 [31] 则使用了更多不同类型的资源使用数据, 包括每个组件实时的用户 CPU 占用率 (CPU_user)、系统 CPU 占用率 (CPU_system)、内存使用量 (mem_used)、每秒读取硬盘扇区的数量 (rd_sec/s)、每秒写入硬盘扇区的数量 (wr_sec/s)、每秒收到的网络包的数量 (rxpck/s)、每秒发出的网络包的数量 (txpck/s) 以及实时的 TCP 和 UDP 连接数量, 其方法首先对每个监控指标在每个时刻的值进行归一化, 并对 TCP/UDP 连接数量的监控数据进行降噪处理, 使用皮尔森积矩相关系数 (Pearson product-moment correlation coefficient) 方法计算任意两个组件在各个资源使用数据上的相似度. 文献 [31] 同时提出了 IEntropy, 用于衡量不同资源使用数据在衡量两个服务的整体相似度时的权重, 进而使用加权后的各个资源使用数据上的相似度计算两个服务的相似度. 最后提出了一种聚类算法 HiKM (hierarchical and iterative K-means) 根据组件之间的相似度来对组件进行聚类, 聚到同一簇的任意两个组件之间存在依赖关系, 其依赖关系的强弱即两者之间的相似度. 文献 [32] 则提出了一种利用 LSTM (long-short term memory) 神经网络模型, 在多维的资源使用时间序列数据中, 发现不同资源使用数据之间相关性的方法, 从而基于资源之间的相关性, 判断服务之间依赖的方法. 首先从 N 类资源使用数据中, 构建利用 $N-1$ 类的资源使用时间序列数据预测第 N 类资源使用数据的预测模型, 然后提出了一种从收敛后的预测模型

中, 提取出最能预测第 N 类资源使用数据的特征, 从而判断资源使用数据之间的相关性, 并利用学习到的资源使用数据之间的相关性, 判断服务之间的相关性.

文献 [33] 提出了流强度 (flow intensity) 的概念用以表示资源使用数据受请求数量影响的强度, 并观测到在复杂的分布式系统中, 很多节点的流强度指标有很强的相关性, 称为不变量 (invariant). 作者总结了两种流强度指标的不变量的模式, 使用 ARX (autoregressive models with exogenous inputs) 方法从众多的流强度指标中学习出具有线性相关关系的流强度, 并提出了 FullMesh 方法, 在大量时间窗口内的众多流强度指标中挖掘恒定成立的不变量, 这些不变量所表示的服务之间具有依赖关系.

文献 [34] 从微服务系统大量微服务的资源使用数据中自动构建微服务之间的依赖关系图, 首先使用 K-Shape 方法对单个微服务上的所有资源使用的时序数据进行聚类, 以降低参与后续计算的监控指标的数量, 进而在聚类后的每一簇中选择与重心 (centroid) 最接近的单个资源使用的时间序列作为该簇的代表, 然后使用格兰杰因果关系检验 (Granger causality tests) 判断不同服务之间簇的代表之间是否存在因果关系, 当两个服务之间存在至少一对簇的代表具有因果关系时, 认为两个服务之间存在依赖关系.

2.1.3 基于统计指标的服务依赖发现

基于统计指标的服务依赖发现方法利用存在依赖关系的两个服务执行时间差 (delay) 与响应时间 (response time) 存在一定规律的特点, 通过分析两个服务间的执行与响应时间关系, 进而推断两个服务之间的依赖关系.

一部分相关工作通过分析服务响应时间的变化来研究服务之间的调用依赖关系. 文献 [15] 研究了一个典型的在线购物服务的系统中, 如何通过对不同的数据表进行干扰注入并检测相应服务响应时间的变化, 判断服务是否依赖于特定数据表及数据库服务. 其通过在一个 Web 系统中对不同的数据表进行不同时长的锁表操作, 并统计每个服务的响应时间, 对每个服务的平均响应时间与锁表时长进行直线拟合, 如果拟合后的直线斜率 k 大于 0, 则服务依赖于该数据服务中的该数据表, 且斜率表示依赖的强度. 文献 [16] 则在文献 [15] 的基础上引入了更多的干扰类型用以进行注入. 文献 [35] 采用了与文献 [15] 类似的方法, 其通过拦截每个服务在一定时间窗口内的所有网络包, 使其延迟传递一定的时间并监控其他所有服务的响应时间, 根据服务的响应时间是否受影响, 以及响应时间受影响的程度, 来判断每个服务与被拦截网络包的服务依赖关系及强弱. 文献 [22] 基于观察服务之间不同的调用方式产生的调用依赖关系, 其反应在响应时间上的相关性特征不同, 通过学习利用被依赖服务响应时间来预测依赖服务响应时间的模型, 可以判断服务之间是否存在依赖关系, 以及存在的调用依赖关系的类型. 将服务之间的调用关系分为 4 类: 单调依赖 (single dependency)、组合依赖 (composite dependency)、并行依赖 (concurrent dependency) 和分流依赖 (distrbuted dpendency), 分别表示两个服务之间的直接调用关系、一个服务依赖多个服务的串行调用, 一个服务依赖多个服务的并行调用以及一个服务在负载均衡场景下对多个服务的调用. 针对 4 类调用关系, 作者分别分析了被依赖服务响应时间与依赖服务响应时间的关系, 给出了预测模型. 通过利用历史数据训练预测模型, 可以预测某个服务的响应时间, 通过对比预测的响应时间符合哪类调用关系, 可以判断服务之间的调用依赖关系的类型.

另一部分相关工作关注具有依赖关系的两个服务之间其服务执行时间差的相关性. 文献 [9] 基于观察如果两个服务之间存在依赖关系, 则两个服务的延迟分布 (delay distribution)(即两个服务开始执行时的时间差) 不会是随机的, 即存在一个或多个峰值 (spike). 该方法首先根据 10 s 内两个服务被调用的起始时间均值与标准差, 选取合适的延迟粒度, 构造出两个服务之间的延迟分布图, 并使用信号处理方法进行去噪, 当检测到两个服务之间的延迟分布存在一个或多个峰值时, 则认为两个服务之间存在依赖关系. 文献 [8] 分别分析了存在调用依赖关系和逻辑依赖关系的两个服务执行时间差影响的特点, 发现在调用依赖关系中, 延迟依赖服务的开始执行时间会导致被依赖服务的开始执行时间有相应的延迟, 延迟被依赖服务的开始执行时间会导致依赖服务的结束执行时间有相应的延迟, 在逻辑依赖关系中, 延迟依赖服务的开始执行时间对被依赖服务的开始与结束执行时间都没有任何影响, 延迟被依赖服务的开始执行时间则会对依赖服务的开始执行时间产生相应的延迟. 基于上述分析, 文献 [8] 提出了一种服务依赖发现方法 Rippler, 通过延迟传递每个服务的第一个网络包, 观察其他服务的响应时间是否被影响, 判断两个服务之间是否存在依赖关系.

2.2 基于系统日志的服务依赖发现

基于系统日志数据的微服务依赖关系发现利用不同日志数据内容或特征, 发现或推断不同微服务的调用路径、逻辑依赖或关联关系. 根据所依赖的日志内容或特征, 相关研究工作可以分为 3 种: 依据统一标识的服务依赖发现, 基于共现概率的服务依赖发现和基于日志频率的服务依赖发现. 依据统一标识的服务依赖发现假设日志文本中存在对不同微服务的标识信息 (例如 IP 等) 或请求的标识信息 (例如 request ID, block ID 等), 通过解析日志文本, 提取标识信息然后通过表示标识关联不同微服务. 基于共现概率的服务依赖发现假设如果两个微服务输出的一些日志存在频繁共现关系, 则两个微服务之间存在服务依赖. 基于日志频率的服务依赖发现统计连续时间窗口内不同微服务输出的日志频率, 将日志频率作为一个核心指标, 通过挖掘不同微服务的该指标之间的分布关系, 挖掘其中因果和关联关系, 最终获取微服务服务依赖.

2.2.1 依据统一标识的服务依赖发现

依据统一标识的服务依赖发现是基于系统日志数据的微服务依赖发现的主流方法. 本方法假设日志文本中包含能够表征请求的标识信息, 如果两个微服务输出日志的标识信息相同且具有先后序列关系, 则说明两个微服务在请求执行过程中存在调用关系, 即存在依赖关系. 文献 [36] 使用日志中 resource ID 和 request ID 关联不同微服务的日志, 构建请求执行路径. 其原理为包含相同 request ID 的日志序列代表一个请求执行路径, 如果执行路径中的两个连续日志由不同微服务输出, 则代表着两个微服务之间存在依赖关系 (请求连续调用). 文献 [24] 介绍了从 HDFS 日志文本中提取 block ID 和 IP 信息, IP 信息用以发现并标识各个微服务, block ID 用于构建请求执行路径, 并通过关联执行路径中的连续日志, 发现微服务依赖. 在很多情况下, 日志文本中不存在一个特殊标识能够标识一个请求执行路径. 为解决这个问题, 文献 [37,38] 假设日志文本中包含多种 ID 信息, 通过多种 ID 信息串联请求执行路径, 最终发现微服务间的依赖关系. 文献 [39] 的主要贡献在于从系统源代码中找到最关键的 ID, 并最终使用这些 ID 对微服务进行依赖关系发现. 具体而言, 首先通过静态代码分析方法, 挖掘出绝对精确的日志之间的转移关系和日志中的关键标识. 然后, 这些关键标识被用于连接跨越不同组件却属于同一个请求的日志, 进而形成了一个跨服务的完整的以日志为节点的请求执行路径.

2.2.2 基于共现概率的服务依赖发现

基于共现概率的服务依赖发现的核心思想是依据单条日志之间的共现概率, 判断输出日志的服务间的依赖关系. 本方法假设如果不同微服务输出的两条日志之间存在着频繁先后共现关系, 则说明两个微服务可能存在逻辑上的因果或关联关系, 并依据此发现服务间依赖关系. 文献 [40] 通过计算不同微服务输出的日志是否频繁顺序出现, 例如微服务 A 输出的 X 日志在微服务 B 输出的 Y 日志前短时间内出现的概率大于某阈值, 则微服务 B 依赖于微服务 A. 文献 [41] 假设所有微服务的机器时间精确统一, 然后把所有微服务输出的日志按顺序排列, 进而使用 LSH (locality-sensitive-hashing) 算法, 对每一个日志, 计算其最近邻居组并过滤最近邻居组, 确定每一个日志模板的直接后继模板节点. 如果两个微服务输出的日志模板存在直接后继关系, 则两个服务之间存在依赖关系.

2.2.3 基于日志频率的服务依赖发现

基于日志频率的服务依赖发现的核心思想是将日志转换成为数值型的指标, 通过分析指标的分布差异或变化趋势, 发现微服务间的依赖关系. 本方法假设伴随着负载变化, 不同微服务输出的日志数量或频率也随之变化, 如果两个微服务输出的日志数量或频率之间存在相关性, 则说明两个微服务有可能共同协作处理相同请求, 因此两者之间存在一些因果或关联关系, 并依据此发现服务依赖关系. 文献 [42] 将资源指标、流量信息和日志等均转换成为多种信号 (signal), 然后使用 PCA 算法对多种信号进行压缩, 最后尝试将不同服务的信号在时间轴上进行前后偏移, 比对信号之间的关联程度. 文献 [36] 首先将系统日志数据按照时间窗口切分, 然后记录每一个时间窗口内的日志数量, 将日志频率作为一个指标信息, 最后使用 PC 因果推断算法计算每一个微服务输出的日志频率指标之间是否存在因果关系, 如果日志频率指标之间存在因果关系, 则说明微服务之间存在依赖关系.

2.3 基于追踪数据的服务依赖发现

基于追踪数据的服务依赖发现技术以分布式追踪技术作为支撑, 通过分布式追踪技术生成一次服务请求在分

布式软件系统中的请求执行路径, 请求执行路径中的事件之间存在因果关系, 当事件的粒度为方法/服务时, 事件之间的因果关系即方法/服务之间的调用关系[24], 每一个请求执行路径中都包含了部分的服务依赖 (事件之间的因果关系) 信息, 而将多个请求执行路径中的服务依赖信息进行合并便能直接且准确地获取分布式软件系统完整的服务之间的调用依赖信息. 当请求执行路径中事件为细粒度的系统调用、方法调用时, 从请求执行路径中构建服务依赖关系图需要首先对请求执行路径进行抽象, 将细粒度的事件聚合为服务, 然后根据服务之间的因果关系判断服务之间调用依赖关系. 因此, 基于追踪数据发现服务之间的调用依赖关系的核心问题是如何判断事件之间的因果关系. 文献 [24] 将事件之间因果关系的判断方法分为两类: 基于规则的事件因果关系判断和基于统计推断的事件因果关系判断.

基于规则的事件因果关系判断方法利用: 1) 追踪数据中含有的标明事件因果关系的 ID (父子事件 ID); 2) 追踪数据中的时间戳; 3) 事件是否依赖于同一数据的读写, 判定事件之间的因果关系. Dapper[43]是谷歌提出的一个分布式追踪系统, 是利用父子事件 ID 判断事件之间因果关系的典型工作. 其提出了 Span 的概念用以表示方法和服务, 请求执行路径的事件粒度为 Span. Dapper 通过侵入谷歌内部被广泛使用的动态链接库, 包括 RPC 库, 线程库及控制流库 (如函数回调) 等等, 实现请求的全局 ID 和父子事件 ID 随请求执行过程进行传播, 进而可以利用父子事件 ID 准确地判断 Span 之间的因果关系, 即方法/服务之间的调用依赖关系. 文献 [44] 为每一个方法分配一个 MID, 侵入软件系统代码并在每次方法调用时, 将调用者的 MID 传递到被调用者并进行记录, 因此使用 MID 即可判断两个方法之间的因果关系. 文献 [18,19] 将服务的 URI 作为标识服务的 ID, 拦截服务之间的相互调用, 进而判断调用者与被调用者之间的调用依赖关系. 类似的工作还有文献 [45,46]. 在一些分布式追踪相关工作中, 只对请求的全局 RequestID 进行传播与记录, 而没有直接用于判断事件之间的因果关系的 ID, 这种情况下, 通常使用事件的时间戳来判断事件之间的因果关系. 如 Pinpoint[47]侵入 JavaBeans, JSP 和 JSP tags 的实现代码, 在请求执行过程中, 自动为请求分配全局唯一的 ID. 该请求 ID 在网络消息之间的传播是通过在 HTTP 协议的头中添加请求 ID, 在节点内则是通过将请求 ID 存放在 Thread Local Storage 中, 随控制流进行传递, 从而生成全局的请求执行路径. 但同一请求执行路径中的事件之间因果关系则需要通过事件的时间戳进行判断. 其他的分布式追踪技术如 Magpie[48]和 Pivot[49]也采用相同方法判断事件之间的因果关系. 文献 [20] 提出了一种判定同一消息的不同组件/应用的读/写之间的因果关系, 进而发现基于消息中间件的分布式系统的组件/应用之间依赖关系的方法. 其通过侵入消息中间件, 提取不同应用对消息中间件读写消息时的标准协议中的 Message ID 字段, 从而判断消息中间件中同一消息的写入和读出事件之间的因果关系.

基于统计推断的事件因果关系判断方法从历史运行时数据中学习事件之间存在因果关系的特征, 进而在使用学习到的特征判断当前的事件之间的因果关系. MysteryMachine[50]首先根据全局的 RequestID 提取出属于统一请求的所有事件 (日志事件), 并生成一个以各个事件为顶点的全连接图. 其将事件之间的因果关系分为 3 类: Happen-before、Mutual exclusion 和 Pipeline, 然后利用大量同类型的服务请求历史数据, 来对全联接图进行精简, 精简的原理为: 当从任意一个历史数据中可以推断出两个事件之间的一类因果关系不成立时, 从全联接图中删除这条边. 精简之后的图中剩余的边即表示了事件之间的因果关系. 文献 [17] 则从历史数据中学习方法之间的时间上的嵌套关系, 判断方法之间是否存在调用关系. 其拦截每个节点上每个服务调用的起止时间, 并从全局中统计服务 B 的调用起止时间嵌套在服务 A 的调用起止时间的频率, 当频率大于指定阈值时, 认为服务 B 嵌套在服务 A 的调用中, 即服务 A 调用了服务 B, 服务 A 与服务 B 存在调用依赖关系.

虽然请求执行路径中仅直接体现了服务调用依赖关系, 但服务之间的逻辑依赖关系同样可以从请求执行路径中较为直接地获取. 例如在图 4 所示的请求执行路径中, 事件之间的因果关系即服务之间的调用依赖关系, 服务之间的调用顺序可以根据各个服务调用的时间戳决定, 为从请求执行路径中发现 ShipmentService 对 PaymentService 的逻辑依赖关系, 首先需要判断在所有此类请求执行路径中, PaymentService 是否先于 ShipmentSerice 被调用; 进而判断调用 PaymentService 的失效是否会导致 ShipmentService 的调用同样失效, 如果 PaymentService 的失效同样会导致 ShipmentService 的失效 (或者失效的概率超过一定阈值), 那么则可以判断 ShipmentService 与 PaymentService 之间存在逻辑依赖关系.

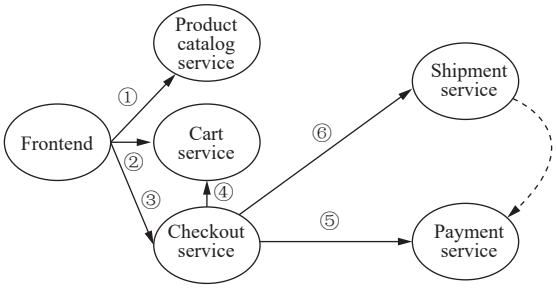


图 4 从请求执行路径中发现逻辑依赖关系

3 服务依赖图的关键应用

服务依赖关系图对于提高微服务架构软件系统的运维效率具有重要意义, 能够帮助开发与运维人员高效、精准地发现与定位系统中的故障并进行根因分析[51−65], 制定高效的资源调度策略[66−72]以保证微服务架构系统的端到端 SLA (service level agreement), 以及用于包括故障预防[73]、部署规划[74,75]和异常检测[76−78]等在内的变更治理. 本节重点介绍基于服务依赖关系图的故障根因定位、资源调度和变更治理等研究工作.

3.1 基于服务依赖图的故障根因定位

随着微服务的发展, 服务间存在复杂的依赖关系, 一个请求往往会引起不同节点上若干服务的大量复杂的互操作. 同时, 这种复杂的依赖关系带来了故障的蔓延性和传播性, 单个服务的故障会在系统中迅速传播. 当某服务出现异常, 可能会级联地导致依赖该服务的其他服务的行为异常, 使得大量的服务失效. 这种跨节点、跨服务的故障传播会大大增加故障根因定位的难度, 降低故障根因定位的效率. 通过构建服务之间的依赖关系图, 在某个服务发生故障时, 可以快速精准地定位故障根因的位置并判断受影响的下游服务. 根据故障根因定位的方法不同, 基于服务依赖关系图的故障根因定位可以分为基于可视化、基于图搜索和基于随机游走方法.

在基于可视化的故障根因定位方面, 文献 [51] 使用可视化工具 ShiViz 对服务依赖关系图进行可视化, 通过人工对比正常追踪数据和异常追踪数据来定位异常根因. 该工作基于异常追踪数据之间具有相似调用链而异常追踪数据与正常追踪数据之间具有不同调用链的假设, 当某个请求发生异常时, 首先从历史数据中收集该请求对应的正常追踪数据和异常追踪数据. 将正常追踪数据和异常追踪数据进行对比, 找到其中不同调用链, 然后将不同异常追踪数据之间进行对比, 找到其中相同调用链. 最后提取两类调用链中的公共部分作为故障根因. 文献 [52] 同样对服务依赖关系图进行可视化, 帮助运维与开发人员理解云服务的架构、行为与状态, 通过人工比对来定位故障根因.

在基于图搜索的故障根因定位方面, 文献 [53] 首先对不同类型的异常设置不同的异常传播方向. 对于性能和可靠性异常, 异常从被调用服务向调用服务传播; 对于流量异常, 异常从调用服务向被调用服务传播. 当检测到异常时, 使用异常发生前给定大小的时间窗口中产生的追踪数据来动态构建服务依赖关系图, 基于服务依赖关系图通过广度优先搜索算法定位根因. 该工作从异常服务出发, 根据异常传播方向访问当前服务的邻居. 对于被访问到的服务, 使用机器学习算法检测该服务是否发生异常, 如果该服务发生异常则纳入异常传播链, 继续对其邻居进行访问, 直到没有新的服务纳入异常传播链. 最后将位于异常传播链末端的服务作为候选根因服务, 并计算候选根因服务与异常服务之间性能指标的皮尔逊相关系数, 根据相关系数对候选根因排序. 文献 [54] 在服务依赖关系图的基础上构建指标因果图, 然后使用深度优先搜索算法沿着指标因果图进行反向遍历来定位故障根因, 对遍历到的指标使用 CUSUM 算法检测异常, 如果当前指标发生异常则继续遍历其子节点, 否则继续遍历其兄弟节点, 每个节点只遍历一次. 当某个异常节点无异常子节点时, 则判断此指标为根因指标. 最后基于 CUSUM 算法计算得到的异常得分对根因指标进行排序. 文献 [55] 的故障根因定位方法与文献 [54] 类似, 但在异常检测上使用 3-σ 准则, 使用根因指标与前端异常指标之间的皮尔逊相关系数作为排序准则. 文献 [56] 首先基于异常发生时间对异常组件

进行排序, 得到异常传播路径, 将异常传播路径的头节点作为故障组件. 然后继续遍历异常传播路径, 如果组件的异常发生时间与头节点的异常发生时间差值较小, 则说明可能是并发故障, 也将其作为异常组件. 最后基于服务依赖图过滤不存在依赖关系的故障组件.

在基于随机游走的故障根因定位方面, 文献 [57] 采用 Personalized PageRank 算法首次将随机游走策略用于定位故障根因. 其基本思想是一个节点 v_j 的指标 m_j 和异常前端节点 v_{fe} 的指标 m_{fe} 的相关性 S_j 表示节点 v_j 是故障根因的可能性. 该工作基于服务依赖关系图构造邻接矩阵, 将相关性 S_j 作为边 e_{ij} 的权重. 由于存在节点本身就是根因而不需要转移到其他邻居节点的情况, 因此将 $\max(0, S_i - \max_{j:e_{ij}\in E} S_j)$ 作为边 e_{ii} 的权重, 即在转移概率中添加自环, 自环的概率代表节点本身就是根因的概率. 此外, 一旦随机游走游走到了错误的路径上, 将无法返回. 因此在邻接矩阵中给每条边添加反向边, 将反向边 e_{ji} 的权重设为 ρS_i. 最后对每个节点的出边根据边的权重进行归一化, 即可得到转移概率. 当异常发生时, 从异常前端节点出发, 根据转移概率从上一个节点的邻居中选择下一个节点进行随机游走, 当所有节点上的 rank 值收敛后, 将节点的 rank 值作为该节点是故障根因的概率. 文献 [58−60] 同样使用 Personalized PageRank 算法在服务依赖关系图上定位故障根因. 不同的是, 文献 [58] 将 Personalized PageRank 算法找到的候选故障根因节点的邻居节点也作为候选故障根因节点. 文献 [59] 使用皮尔逊相关系数计算节点指标间的相关性, 在计算邻接矩阵中边的权重时, 同时考虑节点的相关性和异常得分. 文献 [60] 在服务依赖关系图的基础上添加部署拓扑信息, 并根据边的类型对边设置不同的权重用于故障根因定位. 文献 [61] 在文献 [57] 的基础上, 使用二阶随机游走算法替换 Personalized PageRank 算法, 根据节点被访问的次数对候选故障根因进行排序, 被访问次数越多, 则说明该节点与异常前端节点的相关性越高. 文献 [62,63] 同样基于随机游走算法, 但在计算节点与异常前端节点的相关性时, 考虑了节点之间多指标的相关性. 文献 [64] 首先使用基于统计的异常检测方法计算服务依赖图上节点的异常程度, 然后基于节点之间监控指标的相似度计算节点之间边的权重, 最后基于节点之间边的权重计算异常节点与相邻节点之间的 rank 值, 将 rank 值较低的节点作为根因节点.

服务依赖关系图已在故障根因定位领域得到了大量应用, 服务依赖关系图对服务之间的依赖关系进行刻画, 这种依赖关系反映了服务之间的故障传播, 从而有助于运维人员进行故障根因定位. 基于服务依赖关系图的故障根因定位首先通过指标、系统日志或追踪数据构造服务依赖关系图, 然后当异常发生时, 从服务依赖关系图中的异常节点出发, 通过图搜索、随机游走等算法得到候选故障根因, 然后通过异常分数、与异常节点的相关性或被访问次数等方式对候选故障根因进行排序. 相信未来会有更多研究工作提出结合服务依赖关系图的故障根因定位方法.

3.2 基于服务依赖图的资源调度

基于服务依赖关系图的资源调度可以保证整个服务的 SLA 要求. 文献 [66] 提出一种基于机器学习的微服务资源管理框架. 该框架首先构造用于预测服务端到端延迟和 QoS 违反概率的机器学习模型, 然后使用该模型对资源分配进行预测, 在满足 QoS 的同时最大化资源效率. 文献 [67] 指出资源调度需要考虑微服务之间的依赖性, 否则将导致低效的资源分配, 并且不一定有助于应对负载变化和保证服务性能. 因此, 该工作基于延迟从后端服务传播到前端服务的假设, 利用服务依赖关系图优先对后端微服务进行资源调度, 从而避免前端微服务不必要的资源调度. 文献 [68] 从服务依赖关系图上识别关键路径, 并定位可能违反 SLA 的关键微服务实例, 基于关键微服务实例上的资源利用率、性能指标和工作负载特点, 通过强化学习对资源做出扩容或缩容的决策. 同样地, 基于服务依赖图, 文献 [69] 提出一种基于梯度下降的资源调度算法, 文献 [70] 则使用分层排队网络对微服务的性能建模, 通过遗传算法求解最优资源调度策略.

此外, 一些工作基于服务依赖关系图对虚拟机的部署与迁移进行研究. 文献 [71] 提出一种基于依赖关系的虚拟机部署策略, 将强依赖服务部署到同一物理节点从而降低通信开销. 该工作使用主成分分析对服务依赖关系图进行降维. 然后使用基于引力的聚类算法对虚拟机进行聚类, 从而将在网络流量上具有强依赖关系的虚拟机分到同一组. 最后将虚拟机部署建模为虚拟机组到物理机器的匹配问题, 使用匈牙利匹配算法求解. 文献 [72] 研究了虚拟机迁移顺序问题, 以减少虚拟机迁移带来的服务停机时间. 该工作首先通过虚拟机之间的网络流量得到服务依赖关系图, 然后使用 DFS 算法得到图中的连通子图, 最后将连通子图边的权重设置为流量强度的倒数, 使用最小生成树算法得到同一子图内虚拟机的迁移顺序.

在资源调度领域, 服务依赖图精准刻画了服务/虚拟机之间的依赖关系, 有助于准确地分析服务性能以保证服务端到端 SLA.

3.3 基于服务依赖图的变更治理

在变更故障预防方面, 文献 [73] 基于服务依赖对变更风险进行评估. 首先, 从网络依赖、组件调用等数据构造故障图, 故障图描述了从底层路由器、交换机等网络设备到上层服务的故障传播. 然后, 基于变更通常只会影响小部分服务的假设, 提出一种增量评估算法, 该算法复用已有的评估结果, 将变更风险评估问题转换为布尔可满足性 (SAT) 问题, 使用 SAT 求解器对差异故障图进行分析, 从而避免重新分析整个故障图. 最后, 如果待变更的故障图不满足可靠性目标, 则会根据提供的规则生成一组满足该目标的改进方案.

在变更部署规划方面, 文献 [74] 将服务依赖关系图应用于数据中心中网络设备变更的部署规划, 该工作利用数据中心网络固有的对称性, 在一个大的规划空间中搜索最佳的变更部署规划. 文献 [75] 同样将服务依赖关系图用于网络设备变更的部署规划, 避免由于跨层依赖关系而引起的多个变更实体之间的冲突, 使得最大程度地减少服务中断. 该工作在网络变更前将变更冲突建模为一组约束条件, 将变更部署规划问题转化为优化问题, 使用优化算法求解, 从而得到一组满足所有约束的无冲突的变更部署计划.

在变更异常检测方面, 文献 [76] 将服务依赖关系图应用于网络设备变更后的性能异常检测, 该工作首先基于服务依赖关系图确定变更的影响范围, 然后监控受到变更影响的网络设备的性能指标, 通过 MRLS (multiscale robust local subspace) 算法检测由变更引起的异常. 同样地, 文献 [77] 将服务依赖关系图应用于网络设备变更的影响面识别, 但其考虑了季节性因素、天气和交通流量的变化、网络设备升级等外部因素对变更异常检测的影响. 该工作首先设置研究组 (实施变更的网元) 和控制组 (未实施变更的网元), 提出一种鲁棒的空间回归算法来对比研究组和控制组在变更前后的性能变化, 从而减少外部因素对变更异常检测的影响, 提高研究组异常检测的准确率. 文献 [78] 从系统的角度出发, 设计了一个用于网络设备变更的异常检测系统. 首先该系统基于服务依赖关系图, 将受到变更影响的网络设备作为研究组, 选择研究组地理位置附近未进行变更的网络设备作为控制组. 然后收集研究组和控制组在变更前后的 KPI (key performance indicator) 指标, 对 KPI 指标进行时间对齐、归一化、聚合等操作. 最后使用 MRLS 异常检测算法对研究组检测异常. 文献 [79] 将服务依赖关系图应用于软件变更后的异常检测. 该工作首先根据命名规则得到服务依赖关系图, 基于变更事件和服务依赖关系图进行变更影响面分析, 得到受软件变更影响的服务集合. 然后使用 SST (singular spectrum transform) 算法检测异常. 如果检测出异常, 则根据是否部署软件变更, 将服务分为变更组和对照组, 使用 DiD (difference in difference) 算法判断异常和软件变更之间的相关性.

在变更治理领域, 服务依赖图主要应用于变更影响面的分析, 相信未来会有更多研究工作将服务依赖图应用于变更治理的其他方向.

4 工程实践应用

如前文所述, 微服务系统依赖发现技术可分为基于监控数据的依赖发现, 基于系统日志的依赖发现和基于追踪数据的依赖发现. 目前产业界微服务依赖发现技术往往与数据采集和分析工具紧耦合, 通常作为其中的一个关键功能模块. 日志数据的采集和分析工具包括 Filebeat[80]、Logstash[81]、Flume[82]等开源工具. 其中, 以 ELK (Elasticsearch[83]、Logstash 和 Kibana[84]) 为代表的开源软件生态是业界主流的日志采集存储与分析的技术架构. 此类软件在各个目标节点安装 agent 组件, 读取不同格式的日志数据, 并将数据发送到指定的位置, 收集微服务系统产生的系统日志数据, 然后基于统一标识、共现概率、日志频率等方法预测服务请求执行逻辑, 进而发现微服务实例和微服务依赖关系. Splunk[85]是业界日志数据采集和分析的代表性工具, 支持自定义丰富的日志分析方法, 能够支持构造微服务发现应用.

ZABBIX[86]、Prometheus[87]是当前业界主流的开源监控指标分析工具, 支持资源指标的采集、处理、聚合、可视化等. 基于监控指标分析工具通过收集微服务系统资源指标数据, 分析服务之间网络通信包、资源利用率等数据间的相关性, 进而发现微服务运行时的依赖关系, 从而辅助快速构建微服务依赖发现.

调用链数据刻画了分布式系统中服务请求的完整执行过程. 目前业界主流的分布式追踪工具有 OpenTracing[88]、Zipkin[89]、Jaeger[90]和 SkyWalking[91], DeepFlow[92]等. OpenTracing、Zipkin 及 Jaeger 均源自 Dapper 提出的 Span 的概念, 提出一系列追踪数据的规范与标准, 通过提供不同编程语言的追踪库, 在 Golang、C++等编程语言的系统中通过开发者显式添加追踪代码, 在 Java、Node.js 等编程语言的系统中通过字节码注入、修改 Agent 的方式自动支持追踪, 然后通过统一的数据收集与处理标准, 将调用链数据进行可视化. SkyWalking 在分布式追踪的基础上支持自动化构建服务依赖图并可视化. DeepFlow 则基于 eBPF 技术实现一种更加通用的自动追踪技术. 由于调用链信息天然支持服务依赖发现, 可以精准刻画服务请求的完整执行过程, 构建服务调用链, 进而获取服务之间的依赖关系.

5 服务依赖发现技术展望

表 1 从相关工作所使用的运行时数据的类型、期望发现的服务依赖关系、最终发现的是否为服务运行时数据之间的相关性、是否需要修改目标系统的源代码以及是否需要向软件系统中注入故障或者干扰 5 个角度对比分析了现有的服务依赖发现方法.

表 1 不同服务依赖发现方法对比

运行时数据		文献	目标依赖关系	是否为相关性	是否修改代码	是否注入故障/干扰
监控数据	网络通信包	[5–7,10,11,25,27,35]	逻辑依赖关系	是	否	否
		[12,13,28–30]	调用依赖关系	是		
		[14]	全部	是		
	资源使用	[21]	全部	是	否	是
		[31,32]				否
	统计指标	[8,15,16,35]	全部	是	否	是
		[9,22]				否
系统日志数据		[36–42]	全部	是	否	否
追踪数据		[17]	调用依赖关系	是	是	否
		[18–20]	调用依赖关系	否		

从对相关工作的介绍与表 1 中可以看出, 现有工作通常只能通过挖掘两个服务在其运行时数据上的相关性推断服务之间的依赖关系. 但两个服务具有依赖关系, 是两个微服务的运行时数据具有相关性的充分非必要条件, 即具有依赖关系的两个服务会在不同的运行时数据中体现出相关性, 且相关性的表现可能不止一种, 因此无法根据两个服务在运行时数据中的某种相关性得出两个服务存在依赖关系的结论. 一方面, 这会导致由运行时数据上的相关性推出的服务依赖关系存在较多的误报 (false alarm), 即并非真的服务依赖; 另一方面, 仅依靠一类运行时数据上的一种相关性, 会有较多的漏报 (false negative), 即会忽略掉很多服务之间真实存在的依赖关系. 除此之外, 由于数据中的噪音以及判断相关性中的很多人为的因素 (如需指定时间窗口的大小, 设置模型的超参数, 设置阈值等), 也导致发现的相关性本身就存在较多的误报与漏报. 这进一步导致了从运行时数据的相关性中发现服务依赖关系存在准确率低的问题, 但微服务系统中故障频发、服务依赖极其复杂且故障容易迅速传播导致大量服务失效的特点, 使得在微服务系统中的各类运维任务如根因定位与资源调度对发现的服务依赖关系的准确性有较高的要求. 准确地发现微服务系统中的服务之间的调用依赖关系和逻辑依赖关系, 对保障微服务的可靠性至关重要.

本文从提升服务依赖发现准确率和拓展服务依赖发现技术应用领域两个方面对服务依赖发现技术未来的发展趋势进行展望. 在提升服务依赖发现准确率方面, 针对基于不同类型数据的方法分别给出不同的技术发展趋势. 首先, 基于追踪数据的服务依赖发现方法本身具备精准地发现服务之间的调用依赖关系的能力, 但目前相关技术需要修改目标系统或中间件层的代码, 而微服务软件系统其技术异构的特点使得这种方法难以广泛适用. 因此, 研究通用的分布式追踪技术, 降低在微服务系统中应用分布式追踪技术的难度, 全面、准确刻画请求在复杂的微服务依赖下的执行路径, 从而准确发现服务之间的依赖关系是基于追踪数据的服务依赖发现方法的技术趋势. 其次,

基于日志数据和监控数据的服务依赖发现方法仅能根据数据分布特征模糊地推断依赖关系, 因此, 采用深度学习方法, 将服务依赖发现问题转换为带时序信息的图数据上的链接预测问题, 能够利用标注信息和神经网络的强大表示能力充分捕获数据间的关联关系, 从而提升服务依赖发现准确率, 是基于日志数据和监控数据的服务依赖发现方法的一个未来发展趋势. 最后, 引入额外知识, 如系统文档、配置项数据库信息、专家知识等, 利用知识图谱技术辅助基于追踪数据、日志数据和指标数据发现服务依赖, 弥补数据相关性与真实系统调用依赖和逻辑依赖间的差距, 是提升服务依赖发现准确率的一个未来技术趋势.

在拓展服务依赖发现技术的应用领域方面, 将服务依赖发现技术应用于系统变更风险感知和故障根因定位是一个关键发展趋势. 首先, 服务依赖发现技术能够监测整个变更周期中受变更影响的服务的运行情况, 预测变更可能引起的故障, 实现实时变更风险感知, 帮助运维人员动态调整变更灰度策略, 避免和降低变更造成的故障对业务的影响. 其次, 由于微服务间具有复杂的依赖关系, 发生故障的服务与导致故障的部署了变更的根因服务可能完全不同, 导致故障根因定位十分困难. 针对该问题, 在故障发生时, 基于服务依赖图采用图搜索、因果推断等技术, 可以找到与故障高度相关的变更, 辅助运维人员进行故障根因定位, 提升运维效率.

6 结束语

服务依赖发现技术是准确地刻画微服务架构软件系统中各个微服务之间的复杂的依赖关系的重要手段, 对微服务架构软件系统中的故障定位、性能瓶颈分析、资源调度等一系列运维任务有重要意义, 其研究受到了工业界和学术界的广泛关注.

本文从服务依赖发现的基本概念出发, 从 3 类不同的运行时数据的角度总结了已有的服务依赖发现研究工作. 通过整理总结已有的服务依赖发现技术及其应用的相关工作, 进一步分析了服务依赖发现技术当前所面临的问题并对未来的研究方向进行了展望, 为相关研究人员开展下一步研究工作做出一些有价值的探索.

References:

[1] 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]

[2] IBM. Tivoli. 2021. https://www.ibm.com/docs/en/tivoli-monitoring/6.3.0

[3] Microsoft. Microsoft operations manager. 2006. http://msdn.microsoft.com/en-us/library/aa505337.aspx

[4] Kar G, Keller A, Calo S. Managing application services over service provider networks: Architecture and dependency analysis. In: Proc. of the 2000 IEEE/IFIP Network Operations and Management Symp. The Networked Planet: Management Beyond 2000. Honolulu: IEEE, 2000. 61–74. [doi: 10.1109/NOMS.2000.830375]

[5] Bahl P, Chandra R, Greenberg A, Kandula S, Maltz DA, Zhang M, Claims AI. Towards highly reliable enterprise network services via inference of multi-level dependencies. ACM SIGCOMM Computer Communication Review, 2007, 37(4): 13–24. [doi: 10.1145/1282427.1282383]

[6] Bahl P, Barham P, Black R, Chandra R, Goldszmidt M, Isaacs R, Kandula S, Li L, MacCormick J, Maltz DA, Mortier R, Wawrzoniak M, Zhang M. Discovering dependencies for network management. In: Proc. of the 5th ACM Workshop on Hot Topics in Networks. Irvine: ACM, 2006. 1–6.

[7] Barham P, Black R, Goldszmidt M, Isaacs R, MacCormick J, Mortier R, Simma A. Constellation: Automated discovery of service and host dependencies in networked systems. Technical Report, 2008. 1–14.

[8] Zand A, Vigna G, Kemmerer R, Kruegel C. Rippler: Delay injection for service dependency detection. In: Proc. of the 2014 IEEE INFOCOM-IEEE Conf. on Computer Communications. Toronto: IEEE, 2014. 2157–2165. [doi: 10.1109/INFOCOM.2014.6848158]

[9] Chen X, Zhang M, Mao ZM, Bahl P. Automating network application dependency discovery: Experiences, limitations, and new solutions. In: Proc. of the 8th USENIX Conf. on Operating Systems Design and Implementation. San Diego: USENIX Association, 2008. 117–130.

[10] Kandula S, Chandra R, Katabi D. What 's going on: Learning communication rules in edge networks. In: Proc. of the 2008 ACM SIGCOMM Conf. on Data Communication. Seattle: ACM, 2008. 87–98. [doi: 10.1145/1402958.1402970]

[11] Popa L, Chun BG, Stoica I, Chandrashekar J, Taft N. Macroscope: End-point approach to networked application dependency discovery. In: Proc. of the 5th Int'l Conf. on Emerging Networking Experiments and Technologies. Rome: ACM, 2009. 229–240. [doi: 10.1145/1658939.1658966]

[12] Natarajan A, Ning P, Liu Y, Jajodia S, Hutchinson SE. NSDMiner: Automated discovery of network service dependencies. In: Proc. of the 2012 IEEE INFOCOM. Orlando: IEEE, 2012. 2507–2515. [doi: 10.1109/INFCOM.2012.6195642]

[13] Peddycord III B, Ning P, Jajodia S. On the accurate identification of network service dependencies in distributed systems. In: Proc. of the 26th Int'l Conf. on Large Installation System Administration: Strategies, Tools, and Techniques. San Diego: USENIX Association, 2012. 181–194.

[14] Ding M, Singh V, Zhang YP, Jiang GF. Application dependency discovery using matrix factorization. In: Proc. of the 20th IEEE Int'l Workshop on Quality of Service. Coimbra: IEEE, 2012. 1–4. [doi: 10.1109/IWQoS.2012.6245965]

[15] Brown A, Kar G, Keller A. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In: Proc. of the 2001 IEEE/IFIP Int'l Symp. on Integrated Network Management. Integrated Network Management VII. Integrated Management Strategies for the New Millennium. Seattle: IEEE, 2001. 377–390. [doi: 10.1109/INM.2001.918054]

[16] Bagchi S, Kar G, Hellerstein JL. Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment. In: Proc. of the 12th Int'l Workshop on Distributed Systems. Nancy: INRIA, 2001. 151–164.

[17] Gupta M, Neogi A, Agarwal MK, Kar G. Discovering dynamic dependencies in enterprise environments for problem determination. In: Proc. of the 14th Int'l Workshop on Distributed Systems: Operations and Management. Heidelberg: Springer, 2003. 221–233. [doi: 10.1007/978-3-540-39671-0_24]

[18] Novotny P, Ko BJ, Wolf AL. On-demand discovery of software service dependencies in MANETs. IEEE Trans. on Network and Service Management, 2015, 12(2): 278–292. [doi: 10.1109/TNSM.2015.2410693]

[19] Novotny P, Wolf AL, Ko BJ. Discovering service dependencies in mobile ad hoc networks. In: Proc. of the 2013 IFIP/IEEE Int'l Symp. on Integrated Network Management. Ghent: IEEE, 2013. 527–533.

[20] Wu LJ, Li HW, Cheng YJ, Wu YS, Lin HC. Application dependency tracing for message oriented middleware. In: Proc. of the 16th Asia-Pacific Network Operations and Management Symp. Hsinchu: IEEE, 2014. 1–6. [doi: 10.1109/APNOMS.2014.6996526]

[21] Apte R, Hu LT, Schwan K, Ghosh A. Look who's talking: Discovering dependencies between virtual machines using CPU utilization. In: Proc. of the 2nd USENIX Conf. on Hot Topics in Cloud Computing. Boston: USENIX Association, 2010. 17.

[22] Sangpetch A, Kim HS. VDEP: VM dependency discovery in multi-tier cloud applications. In: Proc. of the 8th IEEE Int'l Conf. on Cloud Computing. New York: IEEE, 2015. 694–701. [doi: 10.1109/CLOUD.2015.97]

[23] Microservices-Demo. 2022. https://github.com/GoogleCloudPlatform/microservices-demo

[24] Yang Y, Li Y, Wu ZH. Survey of state-of-the-art distributed tracing technology. Ruan Jian Xue Bao/Journal of Software, 2020, 31(7): 2019–2039 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

[25] Casalicchio E. Dependencies discovery and analysis in distributed systems. In: Proc. of the 6th Int'l Workshop on Critical Information Infrastructures Security. Lucerne: Springer, 2011. 205–208. [doi: 10.1007/978-3-642-41476-3_18]

[26] WinPcap. 2018. http://www.winpcap.org/

[27] Lange M, Möller R. Time series data mining for network service dependency analysis. In: Proc. of the 2016 Int'l Workshop on Soft Computing Models in Industrial and Environmental Applications Computational Intelligence in Security for Information Systems Conf. Int'l Conf. on European Transnational Education. San Sebastián: Springer, 2016. 584–594. [doi: 10.1007/978-3-319-47364-2_57]

[28] Tsubouchi Y, Furukawa M, Matsumoto R. Transtracer: Socket-based tracing of network dependencies among processes in distributed applications. In: Proc. of the 44th IEEE Annual Computers, Software, and Applications Conf. Madrid: IEEE, 2020. 1206–1211. [doi: 10.1109/COMPSAC48688.2020.00-92]

[29] Carroll TE, Chikkagoudar S, Arthur-Durett K. Impact of network activity levels on the performance of passive network service dependency discovery. In: Proc. of the 2015 MILCOM IEEE Military Communications Conf. Tampa: IEEE, 2015. 1341–1347. [doi: 10.1109/MILCOM.2015.7357631]

[30] EIDefrawy K, Kim T, Sylla P. Automated inference of dependencies of network services and applications via transfer entropy. In: Proc. of the 40th IEEE Annual Computer Software and Applications Conf. Atlanta: IEEE, 2016. 32–37. [doi: 10.1109/COMPSAC.2016.68]

[31] Yin JW, Zhao XK, Tang Y, Zhi C, Chen ZN, Wu ZH. CloudScout: A non-intrusive approach to service dependency discovery. IEEE Trans. on Parallel and Distributed Systems, 2017, 28(5): 1271–1284. [doi: 10.1109/TPDS.2016.2619715]

[32] Shah SY, Yuan ZW, Lu SW, Zerfos P. Dependency analysis of cloud applications for performance monitoring using recurrent neural networks. In: Proc. of the 2017 IEEE Int'l Conf. on Big Data. Boston: IEEE, 2017. 1534–1543. [doi: 10.1109/BigData.2017.8258087]

[33] Jiang GF, Chen HF, Yoshihira K. Efficient and scalable algorithms for inferring likely invariants in distributed systems. IEEE Trans. on Knowledge and Data Engineering, 2007, 19(11): 1508–1523. [doi: 10.1109/TKDE.2007.190648]

[34] Thalheim J, Rodrigues A, Akkus IE, Bhatotia P, Chen RC, Viswanath B, Jiao L, Fetzer C. Sieve: Actionable insights from monitored metrics in distributed systems. In: Proc. of the 18th ACM/IFIP/USENIX Middleware Conf. Las Vegas: ACM, 2017. 14–27. [doi: 10.1145/

3135974.3135977]

[35] Schulz A, Kotson M, Meiners C, Meunier T, O'Gwynn D, Trepagnier P, Weller-Fahy D. Active dependency mapping: A data-driven approach to mapping dependencies in distributed systems. In: Proc. of the 2017 IEEE Int'l Conf. on Information Reuse and Integration. San Diego: IEEE, 2017. 84–91. [doi: 10.1109/IRI.2017.85]

[36] Yuan Y, Anu H, Shi WC, Liang B, Qin B. Learning-based anomaly cause tracing with synthetic analysis of logs from multiple cloud service components. In: Proc. of the 43rd IEEE Annual Computer Software and Applications Conf. Milwaukee: IEEE, 2019. 66–71. [doi: 10.1109/COMPSAC.2019.00019]

[37] Tak BC, Tao S, Yang L, Zhu C, Ruan YP. LOGAN: Problem diagnosis in the cloud using log-based reference models. In: Proc. of the 2016 IEEE Int'l Conf. on Cloud Engineering. Berlin: IEEE, 2016. 62–67. [doi: 10.1109/IC2E.2016.12]

[38] Zhao X, Zhang YL, Lion D, Ullah MF, Luo Y, Yuan D, Stumm M. Lprof: A non-intrusive request flow profiler for distributed systems. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 629–644.

[39] Yin K, Yan M, Xu L, Xu Z, Li Z, Yang D, Zhang XH. Improving log-based anomaly detection with component-aware analysis. In: Proc. of the 2020 IEEE Int'l Conf. on Software Maintenance and Evolution. Adelaide: IEEE, 2020. 667–671. [doi: 10.1109/ICSME46990.2020.00069]

[40] Nandi A, Mandal A, Atreja S, Dasgupta GB, Bhattacharya S. Anomaly detection using program control flow graph mining from execution logs. In: Proc. of the 22nd ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. San Francisco: ACM, 2016. 215–224. [doi: 10.1145/2939672.2939712]

[41] Yu X, Joshi P, Xu JW, Jin GL, Zhang H, Jiang GF. CloudSeer: Workflow monitoring of cloud infrastructures via interleaved logs. ACM SIGARCH Computer Architecture News, 2016, 44(2): 489–502. [doi: 10.1145/2980024.2872407]

[42] Jia T, Chen PF, Yang L, Li Y, Meng FJ, Xu JM. An approach for anomaly diagnosis based on hybrid graph model with logs for distributed services. In: Proc. of the 2017 IEEE Int'l Conf. on Web Services. Honolulu: IEEE, 2017. 25–32. [doi: 10.1109/ICWS.2017.12]

[43] Sigelman BH, Barroso LA, Burrows M, Stephenson P. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report, Dapper-2010-1, 2010.

[44] Mi HB, Wang HM, Zhou YF, Lyu MRT, Cai H. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. on Parallel and Distributed Systems, 2013, 24(6): 1245–1255. [doi: 10.1109/TPDS.2013.21]

[45] Mi HB, Wang HM, Cai H, Zhou YF, Lyu MR, Chen ZB. P-Tracer: Path-based performance profiling in cloud computing systems. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. Izmir: IEEE, 2012. 509–514. [doi: 10.1109/COMPSAC.2012.69]

[46] Yang Y, Wang L, Gu J, Li Y. Transparently capturing execution path of service/job request processing. In: Proc. of the 16th Int'l Conf. on Service-oriented Computing. Hangzhou: Springer, 2018. 879–887. [doi: 10.1007/978-3-030-03596-9_63]

[47] 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. Washington: IEEE, 2002. 595–604. [doi: 10.1109/DSN.2002.1029005]

[48] Barham P, Donnelly A, Isaacs R, Mortier R. Using magpie for request extraction and workload modelling. In: Proc. of the 6th Conf. on Symp. on Operating Systems Design & Implementation. San Francisco: USENIX Association, 2004. 18.

[49] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. ACM Trans. on Computer Systems, 2017, 35(4): 11. [doi: 10.1145/3208104]

[50] 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 Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 217–231.

[51] Zhou X, Peng X, Xie T, Sun J, Ji C, Li WH, Ding D. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. IEEE Trans. on Software Engineering, 2021, 47(2): 243–260. [doi: 10.1109/TSE.2018.2887384]

[52] Guo XF, Peng X, Wang HZ, Li WX, Jiang H, Ding D, Xie T, Su LF. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In: Proc. of the 28th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Sacramento: ACM, 2020. 1387–1397. [doi: 10.1145/3368089.3417066]

[53] Liu DW, He C, Peng X, Lin F, Zhang CX, Gong SF, Li Z, Ou JY, Wu ZS. MicroHECL: High-efficient root cause localization in large-scale microservice systems. In: Proc. of the 43rd IEEE/ACM Int'l Conf. on Software Engineering: Software Engineering in Practice. Madrid: IEEE, 2021. 338–347. [doi: 10.1109/ICSE-SEIP52600.2021.00043]

[54] Chen PF, Qi Y, Hou D. *CauseInfer*: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. IEEE Trans. on Services Computing, 2019, 12(2): 214–230. [doi: 10.1109/TSC.2016.2607739]

[55] Lin JJ, Chen PF, Zheng ZB. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In: Proc. of the 16th Int'l Conf. on Service-oriented Computing. Hangzhou: Springer, 2018. 3–20. [doi: 10.1007/978-3-030-03596-9_1]

[56] Nguyen H, Shen ZM, Tan YM, Gu XH. FChain: Toward black-box online fault localization for cloud systems. In: Proc. of the 33rd IEEE Int'l Conf. on Distributed Computing Systems. Philadelphia: IEEE, 2013. 21–30. [doi: 10.1109/ICDCS.2013.26]

[57] Kim M, Sumbaly R, Shah S. Root cause detection in a service-oriented architecture. ACM SIGMETRICS Performance Evaluation Review, 2013, 41(1): 93–104. [doi: 10.1145/2494232.2465753]

[58] Aggarwal P, Gupta A, Mohapatra P, Nagar S, Mandal A, Wang Q, Paradkar A. Localization of operational faults in cloud applications by mining causal dependencies in logs using golden signals. In: Proc. of the 2020 Int'l Conf. on Service-oriented Computing. Dubai: Springer, 2020. 137–149. [doi: 10.1007/978-3-030-76352-7_17]

[59] Zhang ZK, Li B, Wang J, Liu LQ. AAMR: Automated anomalous microservice ranking in cloud-native environment. In: Proc. of the 33rd Int'l Conf. on Software Engineering and Knowledge Engineering. Pittsburgh: KSI Research Inc., 2021. 86–91.

[60] Wu L, Tordsson J, Elmroth E, Kao O. MicroRCA: Root cause localization of performance issues in microservices. In: Proc. of the NOMS 2020 IEEE/IFIP Network Operations and Management Symp. Budapest: IEEE, 2020. 1–9. [doi: 10.1109/NOMS47738.2020.9110353]

[61] Wang P, Xu JM, Ma M, Lin WL, Pan DS, Wang Y, Chen PF. CloudRanger: Root cause identification for cloud native systems. In: Proc. of the 18th IEEE/ACM Int'l Symp. on Cluster, Cloud and Grid Computing. Washington: IEEE, 2018. 492–502. [doi: 10.1109/CCGRID.2018.00076]

[62] Ma M, Lin WL, Pan DS, Wang P. MS-Rank: Multi-metric and self-adaptive root cause diagnosis for microservice applications. In: Proc. of the 2019 IEEE Int'l Conf. on Web Services. Milan: IEEE, 2019. 60–67. [doi: 10.1109/ICWS.2019.00022]

[63] Ma M, Xu JM, Wang Y, Chen PF, Zhang ZH, Wang P. AutoMAP: Diagnose your microservice-based Web applications automatically. In: Proc. of the 2020 Web Conf. Taipei: ACM, 2020. 246–258. [doi: 10.1145/3366423.3380111]

[64] Kandula S, Mahajan R, Verkaik P, Agarwal S, Padhye J, Bahl P. Detailed diagnosis in enterprise networks. In: Proc. of the 2009 ACM SIGCOMM Conf. on Data Communication. Barcelona: ACM, 2009. 243–254. [doi: 10.1145/1592568.1592597]

[65] Soldani J, Brogi A. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. ACM Computing Surveys, 2023, 55(3): 59. [doi: 10.1145/3501297]

[66] Zhang YQ, Hua WZ, Zhou ZZ, Suh GE, Delimitrou C. Sinan: ML-based and QoS-aware resource management for cloud microservices. In: Proc. of the 26th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Detroit: ACM, 2021. 167–181. [doi: 10.1145/3445814.3446693]

[67] Baarzi AF, Kesidis G. SHOWAR: Right-sizing and efficient scheduling of microservices. In: Proc. of the 2021 ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 427–441. [doi: 10.1145/3472883.3486999]

[68] Qiu HR, Banerjee SS, Jha S, Kalbarczyk ZT, Iyer RK. FIRM: An intelligent fine-grained resource management framework for SLO-oriented microservices. In: Proc. of the 14th USENIX Conf. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 2020. 46.

[69] Mirhosseini A, Elnikety S, Wenisch TF. Parslo: A gradient descent-based approach for near-optimal partial SLO allotment in microservices. In: Proc. of the 2021 ACM Symp. on Cloud Computing. Seattle: ACM, 2021. 442–457. [doi: 10.1145/3472883.3486985]

[70] Gias AU, Casale G, Woodside M. ATOM: Model-driven autoscaling for microservices. In: Proc. of the 39th IEEE Int'l Conf. on Distributed Computing Systems. Dallas: IEEE, 2019. 1994–2004. [doi: 10.1109/ICDCS.2019.00197]

[71] Narantuya J, Ha T, Bae J, Lim H. Dependency analysis based approach for virtual machine placement in software-defined data center. Applied Sciences, 2019, 9(16): 3223. [doi: 10.3390/app9163223]

[72] Narantuya J, Zang HN, Lim H. Service-aware cloud-to-cloud migration of multiple virtual machines. IEEE Access, 2018, 6: 76663–76672. [doi: 10.1109/ACCESS.2018.2882651]

[73] Zhai EN, Chen A, Piskac R, Balakrishnan M, Tian BC, Song B, Zhang HL. Check before you change: Preventing correlated failures in service updates. In: Proc. of the 17th USENIX Conf. on Networked Systems Design and Implementation. Santa Clara: USENIX Association, 2020. 575–589.

[74] Alipourfard O, Gao JQ, Koenig J, Harshaw C, Vahdat A, Yu ML. Risk based planning of network changes in evolving data centers. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 414–429. [doi: 10.1145/3341301.3359664]

[75] de Andrade C, Mahimkar A, Sinha R, Zhang WY, Cire A, Rana G, Ge ZH, Puthenpura S, Yates J, Riding R. Minimizing effort and risk with network change deployment planning. In: Proc. of the 2021 IFIP Networking Conf. (IFIP Networking). Espoo and Helsinki: IEEE, 2021. 1–9. [doi: 10.23919/IFIPNetworking52078.2021.9472779]

[76] Mahimkar A, Ge ZH, Wang J, Yates J, Zhang Y, Emmons J, Huntley B, Stockert M. Rapid detection of maintenance induced changes in service performance. In: Proc. of the 7th Conf. on Emerging Networking Experiments and Technologies. Tokyo: ACM, 2011. 13. [doi: 10.

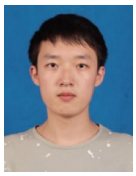
1145/2079296.2079309]
[77] Mahimkar A, Ge ZH, Yates J, Hristov C, Cordaro V, Smith S, Xu J, Stockert M. Robust assessment of changes in cellular networks. In: Proc. of the 9th ACM Conf. on Emerging Networking Experiments and Technologies. Santa Barbara: ACM, 2013. 175–186. [doi: 10.1145/2535372.2535382]
[78] Mahimkar A, Ge ZH, Ahuja S, Pathak S, Shafi N. Rigorous, effortless and timely assessment of cellular network changes. In: Proc. of 49th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Portland: IEEE, 2019. 256–263. [doi: 10.1109/DSN.2019.00037]
[79] Zhang SL, Liu Y, Pei D, Chen Y, Qu XP, Tao SM, Zang Z, Jing XW, Feng M. FUNNEL: Assessing software changes in Web-based services. IEEE Trans. on Services Computing, 2018, 11(1): 34–48. [doi: 10.1109/TSC.2016.2539945]
[80] Filebeat. 2022. https://github.com/elastic/beats
[81] Logstash. 2022. https://github.com/elastic/logstash
[82] Flume. 2022. https://flume.apache.org
[83] Elasticsearch. 2022. https://github.com/elastic/elasticsearch
[84] Kibana. 2022. https://github.com/elastic/kibana
[85] Splunk. 2022. https://www.splunk.com
[86] ZABBIX. 2022. https://www.zabbix.com
[87] Prometheus. 2022. https://prometheus.io
[88] Opentracing. 2022. https://opentracing.io
[89] Zipkin. 2022. https://zipkin.io
[90] Jaeger. 2022. https://www.jaegertracing.io
[91] SkyWalking. 2022. https://skywalking.apache.org
[92] DeepFlow. 2022. https://deepflow.yunshan.net/index.html

附中文参考文献:

[24] 杨勇, 李影, 吴中海. 分布式追踪技术综述. 软件学报, 2020, 31(7): 2019–2039. http://www.jos.org.cn/1000-9825/6047.htm [doi: 10.13328/j.cnki.jos.006047]

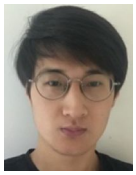
张齐勋(1979－), 男, 博士生, CCF 学生会员, 主要研究领域为软件工程, 智能运维.

贾统(1993－), 男, 助理研究员, CCF 专业会员, 主要研究领域为分布式系统, 智能运维.



吴一凡(1997－), 男, 博士生, CCF 学生会员, 主要研究领域为智能运维, 云计算.

李影(1975－), 女, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为分布式计算, 可信计算.



杨勇(1993－), 男, 博士, 主要研究领域为分布式系统, 云计算, 分布式追踪.

吴中海(1968－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为大数据技术, 系统安全, 嵌入式软件.