软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2020,31(7):2019−2039 [doi: 10.13328/j.cnki.jos.006047]

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

分布式追踪技术综述*

杨 勇[1,2], 李 影[1,2], 吴中海[1,2]

[1](北京大学 软件与微电子学院,北京 102600)

[2](北京大学 软件工程国家工程研究中心,北京 100871)

通讯作者: 李影, E-mail: li.ying@pku.edu.cn

摘 要: 随着分布式软件系统在各个行业的广泛应用,如何提升系统运维效率,保障其服务的可靠与稳定,得到了学术界与工业界的关注.分布式软件系统其规模庞大、结构复杂、持续更新且大量服务请求并发执行的特点,给分布式软件系统的运维任务带来了严峻的挑战.传统的以组件/节点/进程/线程为中心的系统监控与追踪方法难以支持分布式软件的故障诊断、性能调优、系统理解等运维任务.分布式追踪技术识别并提取出分布式软件系统因处理单个服务请求所产生的因果相关的事件,以服务请求为中心对分布式软件系统的行为进行精准、细粒度地刻画,对提高分布式软件系统的运维效率有重要意义.对分布式追踪技术的研究与应用进行了综述,从追踪数据获取、请求事件提取、因果关系判断及请求路径表示这 4 个方面总结了分布式追踪技术的现状;同时以基于请求执行路径的故障诊断和性能分析为例,讨论了学术界对分布式追踪技术的应用研究;最后,对分布式追踪技术的数据读写依赖问题、通用性问题和评价问题进行了探讨并对未来的研究方向进行了展望.

关键词: 分布式追踪;故障诊断;分布式软件系统

中图法分类号: TP311

中文引用格式: 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

英文引用格式: 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). http://www.jos.org.cn/1000-9825/6047.htm

Survey of State-of-the-art Distributed Tracing Technology

YANG Yong[1,2], LI Ying[1,2], WU Zhong-Hai[1,2]

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

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

Abstract: As distributed computing and distributed systems are being widely applied in various areas, how to improve the efficiency of system operations to guarantee the stability and reliability of the services provided by these distributed systems have gained massive momentum from both academia and industry. However, system operation tasks are confronted with tough challenges due the large scale, the intricate structures and dependency, the continuous updating and concurrent service requests of distributed systems. Previous component-/node-/process-/thread-centric monitoring and tracing methods are not sufficient to support the system operation tasks such as fault diagnosis, performance optimization, and system understanding in a distributed system. To address this issue, distributed tracing is proposed and designed. Distributed tracing identifies all the events belonging to the same request and causally correlates these events. Distributed tracing technology precisely and fine-grainedly depicts the behavior of a distributed system in a service-request or workflow-centric way, which is critical to improve the efficiency of system operations. This paper presents a comprehensive survey of existing research work and application of distributed tracing technology. A research framework is proposed and existing research achievements in this field are compared and analyzed with this framework from four perspectives which are

* 基金项目: 广东省重点领域研发计划(2020B010164003)

Foundation item: Key R&D Project of Guangdong Province (2020B010164003)

收稿时间: 2019-05-30; 修改时间: 2019-09-04, 2020-01-18; 采用时间: 2020-03-16; jos 在线出版时间: 2020-04-21

acquiring tracing data, identifying the events from the same request, determining the causal relationships among these events, and representing the request execution path. Then the research work of applying distributed tracing technology to system operation tasks such as fault diagnosis and performance optimization is briefly introduced. Finally, the data dependency issue, the generality issue, and evaluation metrics issue of distributed tracing are discussed and a perspective of the future research direction in distributed tracing technology is presented.

Key words: distributed tracing; fault diagnosis; distributed system

分布式软件系统正广泛应用于互联网、金融、政府、工业等社会的各个领域中,对人们的生活有着重要的影响.如何提升系统运维效率,保障分布式软件系统持续提供可靠稳定的服务,成为学术界和工业界共同关注的问题.由于分布式软件系统具有以下特点:(1) 节点规模庞大.分布式软件系统被部署在大量的物理或虚拟节点中;(2) 系统结构与组件依赖关系复杂.系统组件众多且来源多样,可能由不同团队使用不同编程语言开发并集成了第三方或者开源软件代码,组件之间相互依赖;(3) 规模与结构持续更新.为满足不断增长的业务量与新需求,分布式软件系统往往会在规模与结构上持续更新与升级,使得分布式软件系统的规模更加庞大,结构更加复杂.同时,分布式软件系统存在大量并发的服务请求,请求执行的逻辑复杂,即使是单个服务请求也会经过分布在大量节点上的不同组件的处理.这些特点导致了分布式软件系统的行为难以理解,请求执行的异常难以检测与定位,分布式软件系统的性能瓶颈难以分析与优化,给分布式软件系统中的故障诊断、性能调优、系统理解等一系列运维任务带来了严峻的挑战.

图 1 以 Hadoop 系统为例,展示了一个服务请求在分布式软件系统中的处理过程.Hadoop 被部署在由成千上万台服务器组成的大规模集群中,对内或对外提供不同类型的服务(如 HDFS 的文件存取服务,YARN 的资源调度服务,MapReduce 的分布式计算服务等),服务之间相互依赖,不同的服务由不同的组件提供,每个组件有一个或多个分布在不同的物理/虚拟的计算节点上的实例,用户对 Hadoop 提供的服务进行调用(即进行服务请求),所有的组件/服务/节点共同协作响应用户的服务请求.在这种典型的分布式软件系统场景中,传统的以组件/节点/进程/线程为中心的系统监控与追踪方法只能揭示分布式软件系统局部的状态与行为,或者提供全局的粗粒度的统计数据,无法准确刻画分布式软件系统中因服务请求产生的一系列行为与状态变化,难以支持分布式软件系统中的各项运维任务.因此,分布式软件系统迫切需要一种对系统运行行为进行整体、精准、细粒度刻画的技术,满足分布式软件系统的故障诊断、性能调优、系统理解、资源审计等一系列运维管理任务的需求.基于此,分布式追踪技术应运而生.分布式追踪(distributed tracing)是指在服务请求进入分布式软件系统到系统完成对请求的响应这段时间内,将系统中因处理该请求产生的因果相关的事件(即请求事件)相关联,生成请求在系统中的执行路径的过程,也被称为以工作流为中心的追踪(workflow-centric tracing)/端到端追踪(end-to-end tracing)/因果追踪(causal tracing)/分布式调用追踪/全链路追踪[1−3].

分布式追踪技术产生的请求执行路径能够精准地刻画分布式软件系统整体的执行逻辑.基于生成的请求执行路径,开发与运维人员能够高效、精准地发现、定位分布式软件系统的故障并进行根因分析,准确地发现、分析并理解分布式软件系统中的性能瓶颈,在缺乏技术文档的条件下可以理解分布式软件系统的部署结构和运行状态,对系统资源进行细粒度的管理与审计,这对提高分布式软件系统的运维效率具有重要的研究意义.

分布式追踪技术早期与针对单点程序的追踪技术,如 strace、ptrace、DTrace[4],与分布式软件系统中的服务依赖发现[5]密切相关,但针对单点程序的追踪技术只关注一个节点或同一进程/线程的事件之间的时序关系,而服务依赖发现则只关注服务或组件之间的交互.早期对分布式软件系统的追踪主要使用 strace、ptrace、GDB 等,但是,随着分布式软件系统的急速发展与应用,传统的单点调试与追踪工具无法应对规模越来越大、结构越来越复杂、请求并发度越来越高的分布式软件系统,分布式追踪技术逐渐成为一个单独的研究热点,获得了学术界与工业界的大量关注,在工业界的应用中验证了其巨大的商业价值[6].图 2 给出了自 2001 年以来学术界在分布式追踪领域所发表的学术论文的数量、发表源及其发表年限.从图 2 可以看出,自 2001 年来,学术界均有稳定数量的分布式追踪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,且近 6 年来的论文发表数量有明显的增加,成为分布式系统领域一个重要的研究热点.虽然分布式追踪技术在研究与应用方面都取得了

巨大的进展,但当前的分布式追踪技术仍存在着一定的局限性,使得分布式追踪技术无法彻底地发挥其巨大价值,如当前的分布式追踪技术或者需要侵入分布式软件系统的源代码,或者依赖于特定的中间件与特定的链接库,或者在生成的请求执行路径的准确性与完整性上有所缺失.因此,分布式追踪技术在如何低开销、低侵入地生成请求追踪数据,如何从请求追踪数据中构建完整、准确的请求执行路径仍然面临着很大的挑战.本文从追踪数据获取、请求事件提取、因果关系判断及请求路径表示这 4 个方面综述了分布式追踪技术的现状,然后介绍了近年来基于分布式追踪技术的应用研究工作,最后对分布式追踪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 节对分布式追踪相关的概念进行阐述.第 2 节给出研究框架并对分布式追踪领域已有的研究工作进行详细的介绍与分析,并总结其优缺点.第 3 节展示学术界基于分布式追踪技术的应用研究.第 4 节对分布式追踪技术未来的研究方向进行展望.

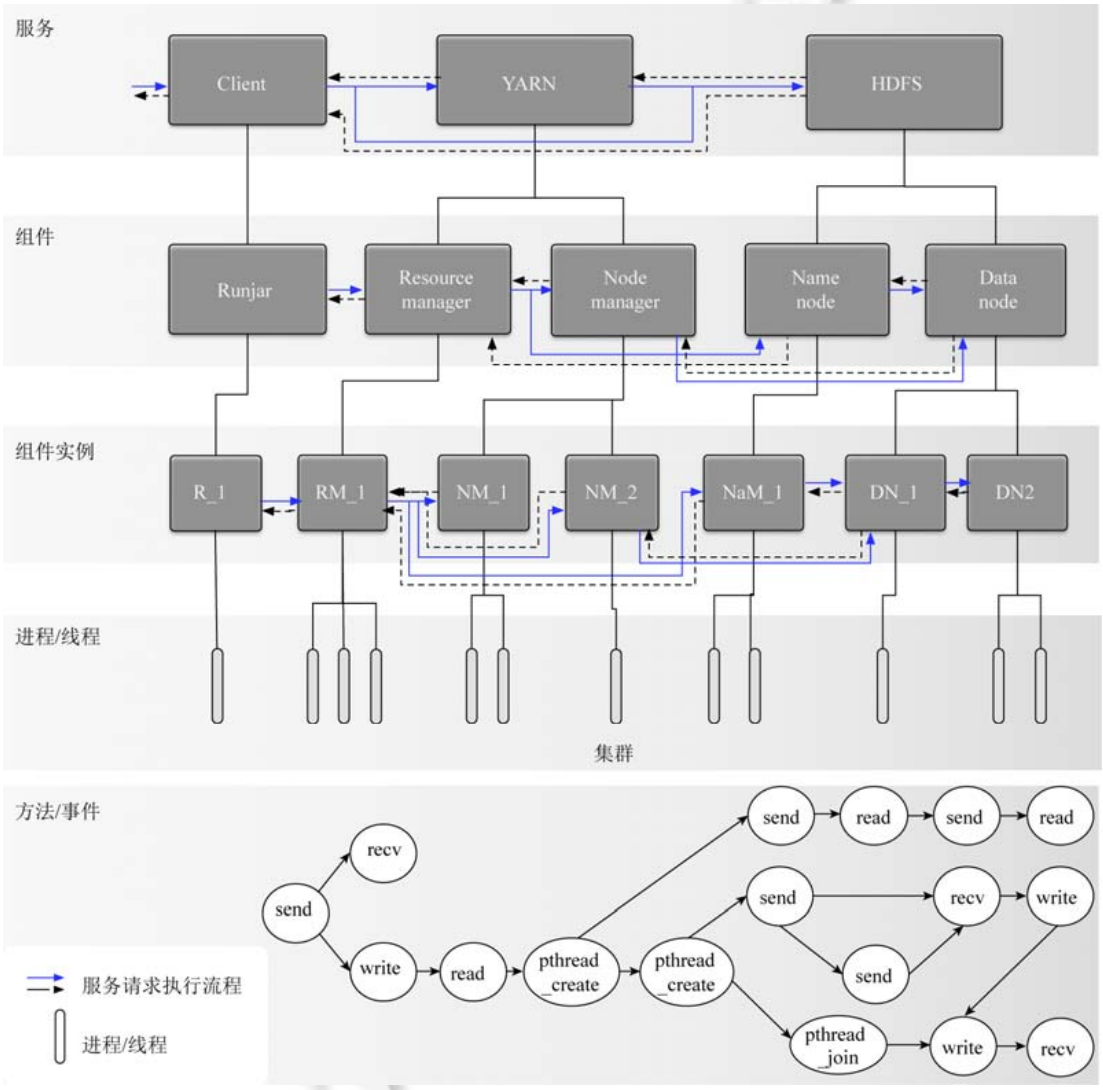

Fig.1 The request processing in a distributed system (Hadoop)

图 1 分布式软件系统中的服务请求处理(以 Hadoop 为例)

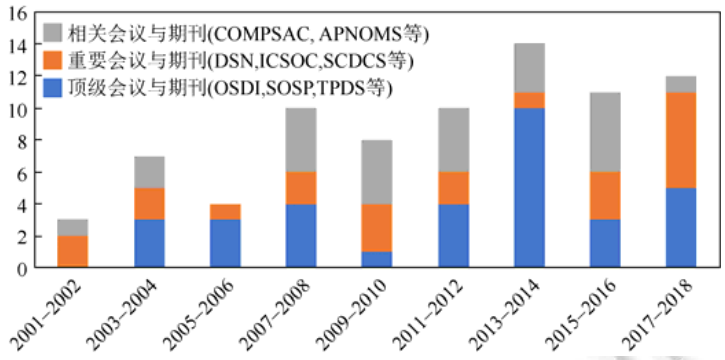

Fig.2 The distribution for the number and the year of the published papers

图 2 论文发表数量、出处及发表年份分布

1 问题描述

假设分布式软件系统在一段时间内处理 n 个服务请求 $R=\{r_1,r_2,r_3,\ldots,r_n\}$,产生的事件的集合为 $E=\{e_1,e_2,e_3,\ldots,e_m\}$,$m\geqslant n$,其中,$r_i$ 为分布式软件系统处理的一个请求,e_i 为一个分布式追踪系统所观察到或产生的事件.

事件:由于不同的分布式追踪技术其所针对的运维任务不同且其所在的软件堆栈也不同,所以在不同的分布式追踪研究工作中,其事件的粒度(granularity)可能是不同的,常见的事件粒度有内核事件、系统调用事件、库调用事件、方法调用事件、RPC(remote procedure call)事件、日志消息、组件及服务等.

因果关系:事件之间的因果关系通常被定义为 Lamport[7]提出的事件之间的 Happen-before 关系.Happen-before 关系定义事件 $a\rightarrow b$,即事件 a 为事件 b 的因,事件 b 为 a 的果,当且仅当:(1) 若 a 与 b 在同一进程(可扩展至子进程及线程之中)中,a 先于 b 发生;(2) 若 a 是一个消息的发送事件,而 b 是此消息的接收事件.相关工作对 Happen-before 关系进行了拓展,如 Yong[1]在 Happen-before 因果关系的基础上拓展出因进程/线程同步导致的事件之间的因果关系及因读写同一内存地址或消息队列等产生读取与写入事件之间的依赖关系,文献[8]将 Log 片段之间的因果关系划分为 Happen-before、Mutual exclusion 和 Pipeline 这 3 类.若事件粒度为方法调用事件,因果关系则通常指的是方法之间调用与被调用的关系[9].若事件粒度在组件及服务级别,因果关系则通常指的是组件之间的调用关系[10,11],或服务之间的依赖关系[12].理想状态下,分布式追踪技术应该能够识别所有事件之间的因果关系.

请求执行路径:获取事件集合 E,将 E 划分为 n+1 个子集合 $E_1,E_2,\ldots,E_{n+1}$,判断每个子集合 E_j 中事件之间因果关系并对每一个请求构建请求执行路径.其中,当 $i\neq j$ 时,$E_i\cap E_j=\varnothing$,且 $E_1\cup E_2\cup E_3\cup\ldots\cup E_{n+1}=E$,其中,$E_{n+1}$ 为分布式软件系统自身产生的与服务请求无关的事件集合;每个子集合 $E_j=\{e_1^j,e_2^j,e_3^j,\ldots,e_o^j\}$,$o$ 为集合中事件的个数,集合中的每个事件 $e_i^j\in E$,且 e_i^j 是分布式软件系统因处理请求 r_j 过程中产生的事件;对每个子集合 E_j,分布式追踪生成该集合中事件之间的因果关系集合 $C_j=\{c_1^j,c_2^j,c_3^j,\ldots,e_p^j\}$,$p$ 为集合中因果关系的个数,其中,每个因果关系 c_t^j 表示 E_j 中的两个事件 e_i^j 与 e_k^j 存在因果关系即 $e_i^j\rightarrow e_k^j$;对每一个请求 r_j,分布式追踪系统根据其事件集合 E_j 与其因果关系集合 C_j 构建请求执行路径.

2 分布式追踪技术相关研究工作

2.1 技术框架

分布式追踪技术的典型框架如图 3 所示,由 4 个关键技术组成:(1) 追踪数据获取;(2) 请求事件提取;(3) 事件因果关系判断;(4) 请求执行路径表示.

(1) 追踪数据获取.在系统运行时获取构建请求执行路径所需的追踪数据.追踪数据包括事件本身与辅助构建请求执行路径的其他数据.例如,当请求执行路径中的事件为方法调用时,请求执行路径为一个请求的方法

调用树,追踪数据包含每个方法调用事件、请求 ID、每个方法调用事件的父事件 ID、每个方法调用的时间戳与执行时间等.追踪数据获取主要有两个技术方向:① 代码侵入;② 数据收集.前者通过在软件堆栈的不同层次修改代码设置追踪点生成追踪数据,后者利用系统运行时产生的日志与内核事件作为追踪数据.

(2) 请求事件提取.从所有事件中将与某一请求相关的事件提取或隔离出来.例如,当请求路径中的事件粒度为方法时,请求事件提取的是系统为处理该请求所调用的所有方法的事件集合.请求事件提取主要有 3 个技术方向:① 基于请求标识;② 基于统计推断;③ 基于因果关系判断.基于请求标识的请求事件提取方法利用请求的全局 ID 或者局部 ID 判断事件是否属于同一个请求,基于统计推断的事件提取方法采用统计学方法推断事件是否属于同一个请求,基于因果关系判断的事件提取方法根据事件之间的因果关系不断地将事件连接,相互连接的事件属于同一请求.

(3) 因果关系判断.对属于某一请求或所有的事件进行因果关系判定,若两个事件存在因果关系(*A* 为因,*B* 为果),则事件 *A* 是事件 *B* 发生的前提.例如,当请求执行路径中的事件粒度为方法时,该技术判断属于同一请求的任意两个方法调用事件之间的调用关系.因果关系判断主要有两种方法:① 基于规则;② 基于统计分析.前者利用追踪数据中所含有的父子事件 ID、时间戳、对同一数据的读写关系等判断事件之间的因果关系.后者采用统计学方法,从历史或者全局的追踪数据中,推断事件之间的因果关系.

(4) 请求执行路径表示.对请求的事件及事件之间的因果关系进行表示.例如,当请求执行路径中的事件粒度为方法时,请求执行路径可以表示为一个请求的方法调用序列或者方法调用树.请求执行路径主要有 3 种表示模型:① 序列模型;② 树模型;③ 有向图模型.

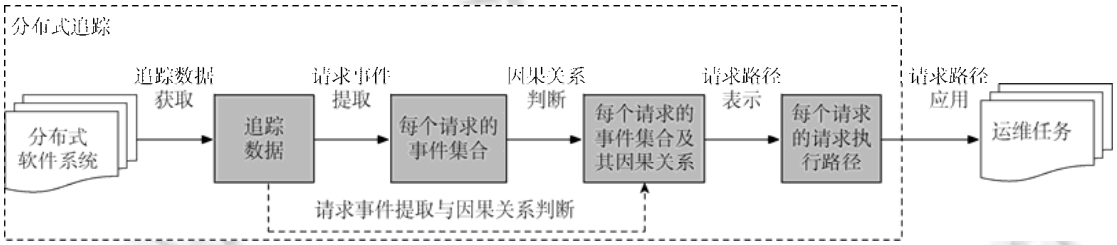

Fig.3 The research framework of distributed tracing technique

图 3 分布式追踪技术框架

2.2 追踪数据获取

追踪数据包括两部分:(1) 构成请求执行路径的事件,如内核事件、方法调用事件等;(2) 辅助构建请求执行路径的其他数据,如时间戳、请求 ID 等.根据是否可以控制所获取的追踪数据的内容,将追踪数据获取技术分为基于代码侵入的追踪数据获取和基于数据收集的追踪数据获取.前者采用侵入软件堆栈的不同层次的方式主动获取构建请求执行路径所需要的追踪数据,后者则收集分布式软件系统在运行过程中产生的预定义的数据.

2.2.1 基于代码侵入的追踪数据获取技术

此类技术主要研究如何以较少的代码修改、较低的系统开销及较强的通用性侵入软件堆栈的不同层次,设置追踪点,在分布式系统软件中传播请求的上下文,实现追踪数据的获取.各技术的主要区别在于侵入软件堆栈的层次、侵入的方法、追踪点的设置及请求上下文的传播机制.按照侵入软件堆栈层次的不同,我们可以将基于代码侵入的追踪数据获取技术更具体地划分为以下 4 类.

(1) 侵入分布式软件系统.此类方法需要人工修改分布式软件系统的源代码,添加支持分布式追踪功能的代码,产生请求的标识符并控制请求的标识符在分布式软件系统中的传播,生成追踪数据.文献[3]在分布式存储系统 Ursa Minor 中,设置了 200 个追踪点,增加了 250 000 行代码用于产生追踪数据.其追踪数据由时间戳、请求的全局 ID——breadcrumb、进程 ID 和线程 ID 所组成的头部以及代表不同事件的负载所组成.其中,请求的全局 ID 在网络消息之间的传播通过修改 Ursa Minor 的 RPC 通信来实现,在节点内则通过人工修改 Ursa Minor 的代码进行传播;负载字段则包含磁盘、CPU、内存等资源使用数据.文献[13]则在文献[3]的基础上增加了部分

追踪点,用于记录多线程编程造成的并行与同步操作.类似的需侵入分布式软件系统代码获取追踪数据的研究工作还包括文献[14−16].

(2) 侵入特定的运行时环境.此类方法需要人工修改分布式软件系统运行时环境的代码或者利用运行时环境的一些特性(如,面向切面编程 AOP)及特定的工具进行半自动化的侵入.这种方法常见于追踪运行在 JAVA 或者.NET 平台之上的分布式软件系统.文献[17,18]侵入 JavaBeans、JSP 和 JSP tags 的实现代码,在请求进入 J2EE 系统时,自动地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,在节点之内则是通过将请求 ID 存放在 Thread Local Storage 中,随控制流而传播.文献[19,20]虽没有侵入 JVM 实现,但依赖于 Java 平台支持面向切面编程(AOP)的特性,动态地设置追踪点,生成追踪数据.但其请求 ID 在网络消息之间的传播是通过人工修改分布式软件系统的 RPC(remote procedure call)协议代码,在跨进程/线程交互部分请求 ID 的传播通过使用 AspectJ 进行动态实现,在线程之内的请求 ID 传播与文献[18]相同,是通过 Thread Local Storage 来实现.文献[21,22]则适用于运行在.NET 平台上的分布式软件系统的追踪,其综合利用 Windows 系统中的 Event Tracing for Windows、.NET profiling API、Detours 等不同工具生成包含内核事件、RPC 调用、系统调用、网络交互等不同层次的追踪数据.文献[23]侵入 Node.js 的运行时环境,设置追踪点,动态地产生客户端与分布式软件系统的追踪数据.

(3) 侵入公共链接库.此类方法需要人工修改分布式软件系统所依赖的公共链接库,如 RPC 库、线程库来实现自动或半自动地对基于此类公共链接库的分布式软件系统的追踪.此类方法多见于大型企业中,不同的分布式软件系统依赖于被广泛调用的企业内部的通用链接库.文献[9]通过对谷歌内部的基础 RPC 库、线程控制库和流程控制库(线程池及异步调用)进行少量的代码修改(1 000 行的 C++代码修改与 800 行的 Java 代码修改),设置追踪点,实现了对基于这些公共链接库的分布式软件系统的追踪数据的生成.文献[24]则是基于 Facebook 内部的公共链接库的产生分布式软件系统的追踪数据.而业界有名的开源分布式追踪系统 zipkin[25]与 opentracing[26]采用了与文献[9]相似的机制,提供了多种语言的常见的公共链接库以支持分布式追踪.文献[27,28]通过修改网络链接库向各种网络协议的头部添加请求 ID 后记录网络消息作为追踪数据.文献[29]在分布式软件系统每次进行系统调用及调用结束时分别记录系统调用名称及时间戳作为请求追踪数据.

(4) 侵入操作系统内核接口.此类方法在虚拟机层或者系统调用的 API 上进行系统调用的拦截与篡改来实现自动地对运行在此类操作系统上的分布式软件系统进行追踪.文献[30,31]在虚拟机管理软件(VMM)层拦截系统调用生成追踪数据,包括 socket 的标识符、TCP 四元组(srcIP,srcPort,destIP,destPort)及线程 ID 这些辅助生成请求执行路径的数据.而文献[1,32,33]则通过 LD_PRELOAD 机制拦截 UNIX-like 系统中的系统调用在 libc 库中的 API,当分布式系统进行系统调用时,通过拦截与篡改系统调用的方式生成追踪数据.文献[34]基于 ptrace 拦截系统调用事件,并将系统调用事件和其他来源的日志数据一起作为追踪数据.文献[35]则仅仅处理 TCP 网络通信系统调用事件,记录下分布式软件系统每次网络通信的 TCP 四元组数据.

2.2.2 基于数据收集的追踪数据获取技术

此类技术主要收集分布式软件系统运行时产生的预定义的数据作为追踪数据.这类预定义的数据主要有两种,一种是分布式软件系统运行所产生的在开发阶段预定义的日志,另一种是操作系统层面预定义的内核事件日志.

(1) 收集分布式软件系统日志.分布式软件系统的日志由开发人员在开发时依据个人经验主观设置,不同的分布式软件系统的日志格式通常有所区别,其本身是为了方便开发人员进行系统功能调试,而非为了追踪分布式系统中的请求,其内容、数量、格式与结构通常无法改变.因此需要使用不同的方法对收集到的日志进行预处理,并从中提取请求的全局 ID、局部 ID 及日志之间的关联关系等用于构建请求执行路径.文献[2,8,36−44]均利用分布式软件系统在运行时产生的系统日志作为追踪数据,从中构建请求执行路径.

(2) 收集操作系统内核事件日志.对于操作系统级别的内核事件,不同的事件类型,其格式也会有所区别,相比于分布式软件系统的日志,内核事件更为底层,缺乏上层应用的信息,且难以理解,需要大量的专家知识对每一种类型的内核事件进行解读.因其不包含上层应用的信息,故其变量也不存在请求的全局 ID.文献[45−47]直

接收集运行分布式软件系统的操作系统产生的内核日志作为请求追踪数据,构建请求执行路径.

2.2.3 不同追踪数据获取技术的对比分析

表 1 总结了不同追踪数据获取技术的优缺点.基于代码侵入的追踪数据获取技术具有追踪内容可自定义、可灵活设置追踪点的优点,即产生什么类型的数据以及在什么位置产生追踪数据乃至于产生的追踪数据的内容是什么都可以进行控制.其缺点则在于需要人工修改相应软件堆栈的代码,且会引入一定的额外的系统开销.侵入分布式软件系统的方法适用于分布式软件系统的源代码可控且对源码比较了解,且不需要考虑后续通用性的情况;侵入运行时环境的方法和侵入公共链接库的方法适用于需要考虑分布式追踪技术的通用性,且被追踪的分布式软件系统具有相同的运行时环境或公共链接库的情况;侵入操作系统内核接口的方法适用于对分布式软件系统的运行环境具有完全的控制(Root 权限),且需要分布式追踪技术具有较高的通用性的情况.而基于数据收集的技术具有无额外系统开销、无需任何代码修改的优点,但其缺点也很明显,其可用的数据局限于开发人员在开发时的主观选择或者一些常用的监控指标,如 CPU/内存/磁盘的利用率等.收集分布式软件系统日志的方法适用于无法修改或难以修改分布式软件系统的源码且软件系统有大量日志输出的情况;收集操作系统内核事件日志的方法适用于运行层次较低的分布式软件系统.在基于代码入侵的技术中,随着软件堆栈层次的不断降低,追踪数据内容的可定制性会降低,其通用性会不断增强,但上层应用的信息也越来越少.如直接侵入分布式软件系统代码可以获取丰富的上层应用(分布式软件系统)的信息,但侵入操作系统内核接口则难以获取上层应用的信息.相比于操作系统的内核数据结构,分布式软件系统的日志含有更丰富的与上层应用相关的信息,但其数量较少.内核事件的数量较多,但缺乏上层应用信息,难以理解.

Table 1 Pros & cons of tracing data acquiring methods

表 1 不同追踪数据获取技术的优缺点

		优点	缺点
基于代码侵入	侵入分布式软件系统	可灵活设置追踪数据的内容	通用性差,需修改大量代码
	侵入运行时环境	无需修改分布式软件系统代码	通用性较差
	侵入公共链接库	无需修改分布式软件系统代码	通用性较差
	侵入操作系统内核接口	无需修改分布式软件系统代码,通用性较高	获取的追踪数据过于底层, 缺乏有价值的上层应用信息
基于数据收集	收集分布式软件系统日志	无需修改任何代码,不引入额外系统开销	可获取的日志依赖于开发人员的主观选择
	收集操作系统内核事件日志	无需修改任何代码,不引入额外系统开销	获取的追踪数据过于底层, 缺乏有价值的上层应用信息, 可能无法生成完整的请求执行路径

2.3 请求事件提取

请求事件提取技术将与某一请求相关的所有事件从获取的所有追踪数据中提取出来,生成以请求为单位的事件集合.其主要解决的问题是,如何将与某一请求相关的所有事件从因请求并发造成的不同请求的事件相互交叉的事件集合中提取出来.请求事件提取技术可进一步划分为:(1) 基于请求标识的请求事件提取,通过追踪数据中含有的请求的全局 ID 或者请求的局部 ID(如请求在组件内的 ID/进程/线程/关键变量等),将与一个请求相关的所有事件从大量追踪数据中提取/隔离出来;(2) 基于统计推断的请求事件提取,通过对分布式软件系统的代码或追踪数据的统计与分析发现与一个请求相关的事件在时间或概率上的规律,根据学习到的规律将事件划分到一个请求当中;(3) 基于因果关系的请求事件提取,首先进行因果关系判断,利用得到的所有事件的因果关系将因果相关的事件关联,相互关联的事件属于同一个请求.

2.3.1 基于请求标识的请求事件提取

此类技术利用追踪数据中可在全局唯一标识一个请求的全局 ID 或在一定范围内(组件、进程、线程等)唯一标识一个请求的局部ID,作为区分不同请求的事件的标识.基于请求标识的方法根据所使用ID的作用范围可进一步划分为以下两类.

(1) 基于请求全局 ID 的请求事件提取方法.该类方法要求获取的追踪数据中每个事件都有一个全局范围内唯一标识一个请求的 ID,请求 ID 相同的事件属于同一个请求.使用此类方法进行请求事件提取有两种途径:

① 在追踪数据获取时,使用代码侵入的方法在产生追踪数据时为一个请求的所有追踪数据标明其所属的请求的全局 ID;② 分布式软件系统产生的日志中存在能够全局唯一标识一个请求的 ID,且每条日志中都含有其所属的请求的全局 ID.文献[3,14,15,18−20,22−24,27,28,48,49]是通过侵入分布式软件系统不同软件堆栈的方式产生请求的全局 ID 并随请求在分布式软件系统中的执行进行 ID 的传播,因此,其所有的追踪数据中都含有请求的全局 ID,将含有同一请求全局 ID 的所有事件聚集,即可获得一个请求的所有的事件的集合.而文献[8]则是利用在 Facebook 社交服务的系统日志中的全局 ID 将每一个日志事件划分至一个请求的事件集合中,这需要开发阶段在分布式系统的源代码中添加对分布式追踪的支持.

(2) 基于请求局部 ID 的请求事件提取方法.此类方法的基本步骤如下:① 基于组件/进程/线程/方法范围内的局部 ID,如请求的局部 ID、线程 ID、日志中的变量等,将一个请求的所有事件根据这些 ID 划分为多个子集合;② 寻找能够关联两个不同集合的事件(即边界事件);③ 合并根据边界事件能够进行关联的两个事件子集合;④ 重复步骤②与步骤③,直至无法寻找到新的边界事件.理想情况下,经过这 4 步后,根据局部 ID 划分的一个请求的多个子集合可以合并为一个集合,这个集合包含与此请求相关的所有事件.如文献[46]收集分布式软件系统运行时操作系统产生的内核事件.其首先将所有内核事件按照线程 ID 划分为多个 Event Slice,将与一个请求相关的所有内核事件划分到多个以线程 ID 为局部 ID 的子集合中.根据对内核事件的分析,寻找可以将两个不同线程 ID 代表的不同的 Even Slice 连接起来的 Marker 事件.当发现 Marker 事件之后,根据 Marker 事件将两个不同的 Event Slice 进行连接生成新的 Event Slice 并继续寻找 Marker 事件,以关联更多的 Event Slice.当无法继续找到 Marker 事件之后,一个请求的所有的内核事件就都被连接到了一个 Event Slice 中,即 Event Sketch.文献[47,50]的请求事件提取方法与文献[46]相似.文献[38]认为,在日志中通常不存在一个请求的全局 ID,但会存在多个与一个请求相关的局部 ID.该方法首先根据网络请求入口找到请求初始的局部 ID,然后将含有此局部 ID 的 Log entries 都提取出来,再从这些 Log entries 中寻找其他可能的局部 ID 以关联更多的 Log entries,如此递归,直至无法关联更多的 Log entries.这些所有关联起来的 Log entries 即提取到的请求事件的集合.文献[37]中的 Object ID、文献[51]中的 UUID 及文献[52]中的 TaskID 均作为类似的局部 ID 的机制来提取属于同一请求的事件.

2.3.2 基于统计推断的请求事件提取

此类技术通过对程序代码或者追踪数据的统计分析发现与一个请求相关的事件在时间或概率上的规律,根据统计规律将事件划分到一个请求当中.根据所使用的分析方法的不同,我们可以将基于统计分析的请求事件提取技术更具体地划分为以下两类.

(1) 基于运行时数据分析的请求事件提取.这类方法使用典型的机器学习方法,在离线学习阶段从系统运行时产生的数据中挖掘同一请求的不同类型事件的关联规则;在在线应用阶段,当出现的事件满足学习到的关联规则时,则将这些事件划分到同一请求的事件集合当中.文献[45]提出了一种基于操作系统的内核事件数据推断请求在系统中的执行路径的方法 PInfer.PInfer 首先将内核事件分为两种:成对出现的 Communication event 和其他非成对出现的 Rest event,两个成对出现的 Communication event 组成一个 Communication Event Pair.PInfer 离线地从串行的内核事件中学习 Communication Event Pair 的检测模型与 Communication Event Pair 之间的转移模型(关联关系).针对前者,提出了一种基于图的 Communication Event Pair 检测算法.针对后者,提出了一种基于马尔可夫过程的计算 Communication Event Pair 之间的转移概率的方法.在在线应用阶段,提出了一种提取请求事件并生成请求执行路径的 Graph-based Communication Path-generation 算法,该算法利用在离线学习阶段学习到的 Communication Event Pair 检测模型和 Communication Event Pair 转移模型提取属于一个请求的所有的事件.其过程如下:根据线程将所有的内核事件进行关联,包括 rest event 和 communication event;根据 Communication Event Pair 检测模型发现所有的 Communication Event Pair;根据 Communication Event Pair 之间的转移模型,选择转移概率最高的 Communication Event Pair 与其关联;根据 Communication Event Pair 以及 Communication Event Pair 之间的转移关系,关联不同线程/节点上的内核事件,从而将与一个请求相关的所有事件提取出来.文献[36]提出了一种利用 LSTM(long short-term memory)深度神经网络从分布式软件系统的日志

中学习异常检测模型,并利用学习到的异常检测模型将属于不同请求的日志事件区分的方法.

(2) 基于静态代码分析的请求事件提取.这类方法从分布式软件系统源代码中发现属于同一请求的日志模板集合并提取可以用于辅助区分请求的关键变量,在系统运行时,将收集到的日志与挖掘到的日志模板进行匹配,结合关联变量列表,将属于同一请求的所有日志提取出来.文献[40]提取一个请求的所有事件的步骤如下:① 通过对 Java 字节码进行反编译获取程序源码,将源码中的日志打印语句进行格式转换,提取出其中的变量与非变量部分,并将其表示为 log point(正则表达式);② 判断每个包含 log point 的方法 *M* 对步骤①所提取到的任何变量是否会进行修改,如果某一个变量被 *M* 以及 *M* 所调用的方法修改,则将其放入 *M* 可修改的变量集 MV(modified variable set)中,否则,将此变量放入 *M* 的请求标识符集合 RIC(request identifier candidate set)中;③ 通过静态代码分析寻找请求的入口方法,一个方法 *M* 是入口方法的标准是其调用者 *N* 的 *RIC* 比 *M* 的 *RIC* 数量要少.经过这 3 步之后,就生成了一个请求执行过程所调用的所有方法以及涉及的 log point 的集合和一个关键变量集合 *RIC*.在系统运行时,系统产生的日志与挖掘到的一个请求执行过程中所涉及到的 log point 集合进行匹配,并根据挖掘到的关键变量集合 *RIC* 区分不同的请求,从而将与某一请求相关的所有日志事件提取至该请求的事件集合中.

2.3.3 基于因果关系的请求事件提取

此类技术依赖于事件之间的因果判断技术,从请求的第 1 个事件起,根据两个事件之间的因果关系不断地关联事件,直至无法关联更多的事件,这些相互关联的事件即属于同一个请求的所有事件.

文献[1]提出了基于 NETCOM_ID 与 DATA_ID 的判断分布式软件系统的系统调用事件之间的因果关系方法,将接收外部服务请求的 *recv* 事件作为请求的第 1 个事件,根据 NETCOM_ID 和 DATA_ID 判断分布式软件系统其他系统调用相互之间的因果关系及与 *recv* 事件的因果关系,将属于该请求的所有因果相关的系统调用事件相关联,从而提取出一个请求的所有事件.文献[9]根据推断的系统调用的事件之间的因果关系,不断地串连与一个请求相关的所有事件.文献[53]使用机器学习方法挖掘事件因果关系的规则,并根据挖掘到的规则判断事件之间的因果关系,根据事件之间的因果关系关联并提取出与一个请求相关的所有事件.

2.3.4 不同请求事件提取技术的对比分析

表 2 总结了不同请求事件提取技术的优缺点.基于请求标识的请求事件提取相比于基于统计推断的请求事件提取,其准确性较高,提取效率也较高,且不需要大量的历史数据或源代码进行离线训练,缺点是依赖追踪数据中的请求ID,或者需要在基于代码侵入获取追踪数据时控制请求ID的生成与传播,或者需要在开发阶段向日志中添加此类 ID.基于请求全局 ID 的方法适用于分布式软件系统会自动地为每个请求产生一个 ID 并随请求的处理而对请求的ID进行传播的情况;基于请求局部ID的方法适用于分布式软件系统对请求在局部的处理过程中会产生一个局部的ID且不同的局部ID之间可以建立关联关系的情况.基于统计推断的请求事件提取不要求追踪数据中含有请求的 ID,但其依赖于特定的假设,如假设同一请求的事件存在时间序列上的模式或者特定的关联规则.当该假设在某些分布式软件系统中不成立时,则无法提取出属于同一请求的所有事件,且通常无法准确提取一个请求的所有事件.基于运行时数据分析的方法适用于不存在请求全局ID或者局部ID且对请求事件提取的准确率要求不高的情况;基于静态代码分析的方法适用于分布式软件系统的源代码可控且网络通信较少的情况.基于因果关系的请求事件提取方法,其优点依赖于因果关系判断技术,在存在大量并发请求的分布式软件系统中,需要判断所有请求的任意两个事件之间的因果关系,其提取的效率会相对较低,适用于对事件提取效率与准确率要求不高的情况.相比于局部 ID,全局 ID 对追踪数据获取技术的要求更高,需要在侵入不同软件堆栈时产生并控制请求的全局 ID 的传播;或者在开发过程中,添加支持分布式追踪的代码,控制请求 ID 的生成与传播,并在日志中添加请求的全局 ID.基于静态代码分析的劣势在于其无法关联同一请求中通过网络交互产生的因果相关的事件,且需要访问源代码,但其对于不跨网络的请求事件(特指日志事件)的提取准确性很高.不同的请求事件提取技术也可以进行组合使用.

Table 2 Pros & cons of methods of distinguishing events from different service requests

表 2 不同请求事件提取技术的优缺点

		优点	缺点
基于请求标识	基于请求全局 ID	提取效率高,提取的准确率高	对追踪数据的要求高,需要所有的追踪数据都包含请求全局 ID
	基于请求局部 ID	提取效率较高,提取的准确率较高	容易导致请求执行路径不完整
基于统计推断	基于运行时数据分析	无需在追踪数据中含有全局或者局部 ID	需要历史数据进行训练,且提取效率低,提取的准确率较低
	基于静态代码分析	无需在追踪数据中含有全局或者局部 ID	需要分布式系统的源代码,无法处理请求中的网络通信
基于因果关系	–	无需在追踪数据中含有全局 ID	提取的效率较低

2.4 因果关系判断

事件因果关系判断需要在得到一个请求的事件集合或所有请求的事件集合之后,判断事件集合中所有事件之间的因果关系,可进一步划分为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,前者基于追踪数据中的父子事件 ID、时间戳等数据,利用已知或人为定义的规则判断事件之间的因果关系,后者利用统计分析方法,从大量历史的请求追踪数据中推断事件之间的因果关系.

2.4.1 基于规则的因果关系判断

此类技术通过追踪数据中含有的标明事件因果关系的 ID(父子事件 ID)或利用追踪数据中的时间戳或者根据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,可进一步划分为以下 3 类.

(1) 基于父子事件 ID 的因果关系判断.这类因果关系判断技术的前提是在追踪数据获取时采用主动侵入的方式,生成用于关联父事件和子事件的 ID 进行传播并写入追踪数据中,利用在父事件中标识的子事件的 ID 或者在子事件中标识的父事件 ID,判定两个事件的因果关系(父事件为因,子事件为果).如文献[15]在侵入分布式软件系统时,在每种方法的入口与返回处分别产生追踪数据,追踪数据中包含对该方法进行调用的方法的 MID,根据 MID 可以判断两个方法调用事件之间的因果关系.文献[1]通过 DATA_ID 机制判断网络消息发送与接收事件的因果关系,DATA_ID 相同的一对网络事件,*send* 事件是 *recv* 事件的因.文献[18]则在组件之间相互调用时,记录当前组件调用的组件的 ID,从而判断组件调用事件之间的因果关系.文献[54,55]同样是通过父子事件 ID 来进行事件之间因果关系的判断.

(2) 基于时间戳的因果关系判断.此类方法在一个请求的事件集合中根据每个事件的时间戳判断事件之间的因果关系,时间戳较小的事件是第 1 个时间戳大于其自身时间戳的事件的父事件,即按照事件发生的时间顺序来决定事件之间的因果关系.根据时间戳判定事件的因果关系的方法主要有两种:① 仅仅使用时间戳的大小来判断所有事件的因果关系[49,56];② 使用时间戳的大小判断在一个节点或者线程内的所有事件的因果关系,使用其他信息,如边界事件[38,51]、网络消息的发送/接收[57],来判断不同节点上的事件之间的因果关系.文献[56]首先根据 Task ID 或者 reservation ID 将所有的日志归类到以 task/reservation ID 为基本单位的文件中,完成请求事件提取过程.然后,将日志事件转换为日志模板.最后,完全按照一个 Task ID 所代表的一个日志文件中的所有日志的时间戳大小对日志事件进行排序,前一个事件是后一个事件的父事件,并生成一个有向图来表示所有事件之间的因果关系.相比于文献[56],文献[3]对时间戳的使用更为合理,其仅在一个节点之内按照时间戳的大小判断事件的因果关系.文献[58]在一个组件的内部使用时间戳来判断事件的因果关系,但不同节点上组件的时间戳的比较需要通过一对网络消息的发送和接收事件来进行推断.

(3) 基于数据读写依赖的因果关系判断.这类方法是在消息队列、消息中间件及共享内存中抓取同一消息的写入与读取,进而判断同一消息的写入是否为读取事件的父事件.文献[59]提出了一种检测基于消息中间件的分布式应用的组件/应用之间依赖关系的方法,即判定同一消息的不同组件/应用的读/写之间的因果关系,其工作原理是基于常见的消息中间件协议中的 Message ID 字段的解析.在组件/应用向消息中间件写入消息和读取消息时分别解析消息中的 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.文献[60]则提出了一种共享内存情况下的特定内存区域的消息读写事件因果关系判断的方法.

2.4.2 基于统计推断的因果关系判断

这类技术采用统计方法从大量的历史追踪数据或分布式软件系统的源代码中发现事件间因果关系的模式,从而自动判定同一个请求的两个事件的因果关系,可进一步划分为以下两类.

(1) 基于运行时数据分析的因果关系判断.此类方法将机器学习方法应用到大量的历史运行时数据中,发现事件之间因果关系的模式,从而在新的请求中使用发现的模式判断请求的所有事件之间的因果关系.文献[8]提出了请求事件因果关系判断方法 Mystery Machine.利用日志中含有的请求的全局 ID,将与某一请求相关的所有事件提取出来,并将所有的事件以 segment 为单位进行组织.segment 即⟨*task*,*start_event*,*end_event*⟩这样一个三元组.所有的 segments 之间都假设存在因果关系.Mystery Machine 利用大量同类型的服务请求的历史数据,对这些因果关系进行精简.在同类型服务的历史的追踪数据中,当发现使某一因果关系不成立的 segment 模式时,删除两个 segments 之间的因果关系.不断重复此过程,直至根据历史的同类型服务请求的追踪数据无法找到使得当前的任一因果关系不成立的模式.文献[61]采用了与文献[8]基本相同的方法,但其初始的全联接图是无向图,因此,根据历史数据对无向边进行精简之后,需要根据 V-Structure 原理确定边的方向,即因果关系.文献[53]提出了一种利用无监督机器学习方法根据日志中的时间戳从历史日志中挖掘日志之间因果关系规则的方法,并将挖掘出的事件之间的因果关系规则应用于到在线的请求中,判断事件之间的因果关系.基于运行时数据分析的因果关系判断方法与基于运行时数据分析的请求事件提取方法在使用机器学习方法上并没有本质的区别,两者的区别在于其目的不同,后者关注如何使用机器学习方法从运行时数据中学习事件属于同一请求的模式;前者关注如何使用机器学习方法从运行时数据中学习事件之间存在因果关系的模式.

(2) 基于静态代码分析.此类方法通过分析分布式软件系统的源代码,推断日志打印点在一个请求的执行过程中所出现的逻辑顺序,这种逻辑顺序即在一个请求中两条日志消息所出现的先后顺序(因果关系).文献[40]通过对 Java 字节码的反编译获取分布式软件系统的源代码,进而从程序源代码中提取日志的模板.其获取日志(模板)之间的逻辑顺序的步骤分为两步:① 在同一方法中,根据日志打印语句出现的先后顺序,生成在同一方法中的日志模板之间的逻辑顺序;② 通过方法调用分析,建立跨方法的同一请求执行路径上的日志之间的逻辑顺序.首先寻找入口方法,然后递归地寻找方法调用,最终生成一棵方法调用树.跨方法的日志之间的逻辑顺序根据方法调用树中节点之间的父子关系来进行判断.Caller 方法输出最后一个日志消息是 Callee 方法输出的第 1 个日志消息的父事件.基于静态代码分析的因果关系判断方法与基于静态代码分析的请求事件提取方法的区别在于,后者通过静态代码分析的方法发现代码中的关键变量,并将此类关键变量用作请求的全局 ID 或局部 ID 用于请求事件提取;前者则通过静态代码分析的方法分析方法之间的调用关系,并将方法之间的调用关系映射为运行时方法调用事件之间的因果关系.

2.4.3 不同因果关系判断技术的对比分析

表 3 总结了不同因果关系判断技术的优缺点.基于规则的事件因果关系判断技术其效率更高,不需要训练阶段,且计算量少,直接根据规则即可判断事件之间的因果关系.而基于统计推断的事件因果关系判断方法的优点在于不需要在追踪数据中存在父子事件 ID,多用于基于分布式软件系统的日志构建请求执行路径的方法.基于父子事件 ID 来判断请求当中的事件的因果关系是效率最高且最准确的方法,但其前提是需要对分布式软件系统的软件堆栈的不同层次进行代码侵入,并在请求执行过程中传播父事件与子事件的 ID,以及两个 ID 之间的父子关系,仅适用于每个事件中都存在其父事件或子事件 ID 的情况.基于时间戳的事件因果关系判断提取效率也比较高,其主要缺陷在于无法判断因多线程编程导致的并行与同步问题造成的事件之间的多依赖关系,即一个事件可能有多个父事件或者子事件,而根据时间戳的大小判断事件因果关系无法抓取到这种多依赖关系,适用于不同节点之间的时钟相对同步且对因果关系判断的准确率要求较低的情况.基于数据读写依赖的方法仅适用于判断数据读取事件与数据写入事件之间的因果关系的情况.基于运行时数据分析的方法适用于对请求事件提取的准确率要求不高的情况.基于静态代码分析的事件因果关系判断方法不受节点间时间不同步问题的影响,但这种方法基本无法处理因网络通信造成的不同节点上的事件之间的因果关系,适用于分布式软件系统源代码可控且网络通信较少的情况.在一种分布式追踪技术中可能同时会使用几种不同的因果关系判断方

法来提高因果关系判断的准确性.如文献[1]使用时间戳判断同一线程内的事件的因果关系,同时使用父子事件ID判断网络消息的发送与接收事件的因果关系.

Table 3 Pros & cons of methods of determining causal relationships for events

表 3 不同因果关系判断技术的优缺点

		优点	缺点
基于规则	基于父子事件ID	判断效率高,判断的准确率高	对追踪数据的要求高,需要每个事件都含有其父事件或子事件的ID;产生一定的系统开销
	基于时间戳	判断效率较高	判断的准确率低;不同机器的时间差问题影响因果关系判定
	基于数据读写依赖	能够判断因同一数据的读写产生的因果关系	无法单独作为一个完整的因果提取技术;只适用于特定场景下消息的读写
基于统计推断	基于运行时数据分析	无需在追踪数据中存在父子事件ID	需要历史数据进行训练,提取的准确率较低
	基于静态代码分析	无需在追踪数据中存在父子事件ID	需要分布式系统的源代码,无法准确判断系统运行时生成的网络相关事件的因果关系

2.5 请求路径表示

请求执行路径表示技术对请求所包含的事件和事件之间的因果关系进行建模,其目标在于最大化地保留事件的因果关系以获得对请求执行路径更准确的刻画以及高效地支撑分布式软件系统的不同的运维任务.请求执行路径表示技术按照表示模型的不同可进一步划分为:(1) 基于序列模型的请求执行路径表示;(2) 基于树模型的请求执行路径表示;(3) 基于有向图模型的请求执行路径表示.

2.5.1 基于序列模型的请求执行路径表示

此类技术将请求的执行过程建模为一个事件序列.分布式软件系统的请求处理是高度并行的,多线程、异步网络通信等都使得序列模型无法准确描述如今分布式软件系统的请求处理过程.因此,选择序列模型来对请求的执行路径进行建模大多出于两方面的考虑:(1) 受限于追踪数据,无法构建除序列外的其他模型.如当追踪数据中仅有时间戳信息可用于辅助判断事件之间的因果关系时,通常会选择序列模型;(2) 运维任务相对简单,但对计算速度要求较高.如在请求行为异常检测过程中,将请求执行路径建模为序列,足够发现一些特定类型的异常[22,49,51].也有研究工作采取有一定容错能力的模型来处理将包含大量并行的请求执行路径建模为序列模型所导致的错误,这些方法大多采用概率图模型来表示请求的执行路径,但其本质依然是将一次请求执行产生的事件作为一个序列.如文献[44]将请求执行路径建模为序列,但考虑到用序列表示并行处理请求的问题,从一个请求的多个序列化的执行路径中构建一个马尔可夫过程(Markov process),当任意一次请求的执行路径(序列)在马尔可夫过程的图模型中存在一条转移路径时,认为请求执行路径没有出现异常.

2.5.2 基于树模型的请求执行路径表示

此类技术将请求的执行过程建模为一棵树.树的根节点是请求的起始节点,代表的是请求进入分布式软件系统所引发的第 1 事件,每一个叶子节点都与其直接父节点存在因果关系(父节点为因,子节点为果).虽然树模型能够表示分布式软件系统中的并行处理,但与多线程并行伴随的往往是线程同步等操作,如 *pthread_join* 等.因此,采用树模型表示并行处理的请求,或者因为追踪技术无法捕获线程及进程的同步操作,或者出于简化分布式软件系统的并行处理的需求,不考虑因进程及线程同步导致的事件之间的因果关系.树模型比较常见的用途是建立一个请求的完整的方法调用链[25,26,54,55],或网络通信的调用树(包括普通 *send*/*recv* 与 RPC 调用)[28,35,62,63],文献[3,16,23,33,50]同样使用树模型来表示请求执行路径.

2.5.3 基于有向图模型的请求执行路径表示

此类技术将请求的执行过程建模为一个图,一般是指有向无环图(DAG).图中每个节点是一个事件,而有向边是事件之间的因果关系,一个事件可能会有多个前驱节点(父事件)和多个后继节点(子事件).图模型是在描述请求执行路径时表达能力最强的模型,能够表示并行处理、线程同步等操作.文献[1,64]等均采用有向无环图模型对请求执行路径进行表示.

2.5.4 不同请求执行路径表示技术的对比分析

表 4 总结了不同请求执行路径表示技术的优缺点.总的来说,序列模型通常无法有效地表示分布式软件系统中准确的请求执行路径,需要对其进行容错处理,将其转换为图/概率图模型,适用于不需要请求执行路径精准刻画请求处理过程且对请求执行路径上的计算效率要求较高的情况.树模型具有表达并行处理的能力,但无法表达因线程同步导致的一个事件含有多个父事件的情况,适用于需要同时兼顾请求执行路径精准刻画请求处理过程的能力与计算效率的场景.而图模型具有最强的表达能力,但构建图模型对追踪数据有较高的要求,或者有大量的同类型的服务请求的历史数据,适用于对请求执行路径上的计算效率要求不高的各种情况.

Table 4 Pros & cons of methods of modeling request execution paths

表 4 不同请求执行路径表示技术的优缺点

	优点	缺点
序列模型	存储、计算效率高;易生成	无法表示并行;无法表示由于线程同步等导致的多父节点的场景
树模型	存储、计算效率较高;易生成;能够表示并行	无法表示由于线程同步等导致的多父节点的场景
有向图模型	对请求执行路径的表示能力强,能够表示并行及多父节点的场景	计算效率低,对追踪数据获取及因果关系判断技术要求较高

3 分布式追踪技术应用研究

3.1 分布式追踪技术应用概况

分布式追踪技术已被应用到大量不同的管理任务中,如文献[65–68]利用分布式追踪技术产生的请求执行路径帮助运维与开发人员理解分布式软件系统的结构、行为与状态,文献[69]利用分布式追踪技术对云计算资源的使用进行精细化地计价,文献[70]利用分布式追踪技术追踪系统中电力资源的消耗,文献[71]则利用分布式追踪技术追踪隐私数据的泄漏问题,文献[48]提出了一种以请求执行为单位的资源管理系统.表 5 给出了近年来国内外工业界所提出的分布式追踪技术及其应用领域.分布式追踪技术产生的请求执行路径能够精准地刻画分布式软件系统整体的执行逻辑,因此对请求执行路径进行存储、查询、可视化可以极大地方便运维人员对分布式软件系统中的请求执行状况进行监控,进而提高故障诊断的效率与准确率,帮助运维人员发现并优化请求执行过程中的性能问题.文献[64]提供了一种在请求执行路径上进行相关事件查询和路径抽象的语言,用于故障诊断.学术界也对请求执行路径的可视化提出了不同的方法[39,42,72].

但使用请求执行路径对分布式软件系统中的请求进行监控,依然需要人工根据请求的执行状态或者设置阈值来判断请求执行是否出现故障与性能问题,并在检测出故障后人工进行故障的定位与根因分析,在出现性能问题后人工分析请求执行的性能瓶颈从而进行优化.对于规模庞大、结构复杂、持续更新的分布式软件系统,这仍然是一项艰巨的工作.因此,如何基于请求执行路径,自动化地进行分布式软件系统的故障诊断、性能分析逐渐成为研究热点.本文重点介绍学术界在基于请求执行路径的故障诊断与性能分析方面的相关研究工作.

Table 5 Application of distributed tracing technology

表 5 分布式追踪技术的应用

系统	企业	应用
Dapper & Census	Google	资源监控,故障根因分析
Zipkin	Twitter,Counsera,知乎	服务依赖分析,架构理解,性能监控
Salp	Netflix	依赖分析,关键路径分析
Jaeger	Uber	架构理解,异常检测,服务依赖分析
鹰眼	淘宝	故障定位,监控,异常检测
Hydra	京东	监控,系统行为理解,性能分析
Canopy	Facebook	性能监控与分析

3.2 基于请求执行路径的故障诊断

分布式软件系统由于复杂的结构和庞大的代码量,不可避免地存在软件缺陷,且往往部署在成千上万的虚

拟/物理节点上,运行环境十分复杂.分布式软件系统内部的缺陷与外部的复杂环境导致分布式软件系统频繁地发生故障.这些故障可能由资源竞争、配置错误、软件缺陷、硬件失效等原因导致,在系统运行时则表现为分布式软件系统中的服务请求执行过程出现异常甚至执行失败.分布式软件系统中的故障诊断主要回答 3 个问题:(1) 分布式软件系统中是否出现故障,即异常检测问题;(2) 故障出现在分布式软件系统的什么位置,即故障定位问题;(3) 分布式软件系统中出现故障的原因,即故障根因诊断问题.

请求执行路径精准刻画了分布式软件系统因处理请求产生的一系列行为,因此,基于请求执行路径,可以检测出更细粒度的请求执行过程的异常,继而检测出使用传统的基于监控和基于日志的故障诊断方法检测不到的系统的故障.当系统发生故障时,根据请求执行路径精准地定位故障的位置,诊断故障的根因.基于请求执行路径的故障诊断主要解决 3 个问题:(1) 异常检测,检测分布式软件系统中的请求是否正常执行;(2) 故障定位,在分布式软件系统发生故障之后,找到引起故障的事件/日志/代码片段/组件等;(3) 根因诊断,对故障的根因进行解释.

在异常检测方面,文献[73]在文献[18]的基础上通过侵入 JVM 实现了其分布式追踪系统,利用分布式追踪系统构建的请求执行路径(组件之间的调用树)对所有的请求执行路径构建一个概率上下文无关文法(probabilistic context free grammar).在行为异常检测阶段,计算待检测的请求执行路径在当前概率上下文无关文法模型中生成的概率,当概率小于某阈值时,认为请求执行过程出现了行为异常,系统发生了故障.我们在 TPC-W WIPSo 应用中通过人工注入故障,应用并对比了文献[73]提出的基于概率上下文无关文法的模型检测路径结构异常和基于 HTTP 错误代码检测系统故障两种方法,进而发现在请求执行路径中应用其模型检测系统故障的方法的准确率达到 90%,远远高于基于 HTTP 的错误代码检测系统故障的准确率.文献[56]中请求执行路径为日志事件的序列,通过提取日志的模板,将请求执行路径转化成一个日志模板的有向图(message flow graph,简称 MFG).在行为异常检测阶段,采用其所提出的图的编辑距离计算算法,计算由待检测请求执行路径构建的 MFG 与由正常请求执行路径构建的 MFG 的相似度,当相似度小于某阈值时,认为待检测的请求执行路径中存在行为异常,系统发生了故障.文献[51]从日志消息中构建请求的执行路径,并对每一种服务请求的正常的执行路径构建一个自动机(automaton),在在线检测阶段,将待检测的请求执行路径输入构建的自动机检测请求执行路径是否出现异常.文献[43]与文献[51]类似,不过其从正常的请求执行路径(日志模板序列)中构建一个有限状态自动机(finite state automaton),用于异常检测.文献[74]提出了一种基于贝叶斯方法从请求执行路径中检测其中是否存在慢性故障的方法.

在故障定位方面,文献[38]从分布式软件系统的日志中推断生成请求执行路径,将请求执行路径表示为日志模板的序列,然后利用他们提出的一种日志对齐算法,对比存在故障的请求执行路径和正常的请求执行路径,定位出导致故障的日志模板及其在代码中的位置.文献[56]从日志中生成请求执行路径的碎片,为每一个任务生成一个 MFG(message flow graph)作为请求的执行路径,然后使用基于图编辑距离的算法进行离群点(与其他任务的 MFG 不同的任务)分析,并提取出离群点中特有的日志模板序列,将故障定位至提取出的少量的日志模板中.而文献[75]则利用从日志中推断出的正常的以日志模板为节点的请求执行路径,检测出待检测的请求的执行路径的异常节点,从而发现故障的日志模板,并通过扫描程序的源代码,建立日志模板与日志打印语句之间的映射关系,从而将故障定位至源代码中产生故障日志的日志打印语句.文献[18]则使用层级聚类算法 UPGMA 检测异常的请求执行路径,并通过雅卡尔相似系数(Jaccard similarity coefficient)将故障定位至具体的组件.

在根因诊断方面,文献[30]基于文献[31]提出的分布式追踪技术,在分布式软件系统中进行大量的故障注入实验并收集故障下的请求执行路径,建立故障与请求执行路径之间的关联关系.在发生故障时将待检测的请求执行路径与数据库中的请求执行路径进行相似度匹配,选择相似度最高的几个有故障的请求执行路径,进而对故障根因做出诊断.

分布式追踪技术已在故障诊断领域得到了大量应用,分布式追踪技术产生的请求执行路径能够帮助运维人员更准确地检测出分布式软件系统中的异常行为[73],提高故障定位的粒度[38],并自动诊断故障的根因[30].基于请求执行路径的异常检测通常从正常的请求执行路径中构建图模型(上下文无关文法、有限状态机

等),将待检测的请求执行路径输入构建的图模型中进行检验.基于请求执行路径的故障定位在异常检测方法的基础上,将图上的每个顶点赋予一定的位置属性(如组件、日志打印代码等).当前,基于请求执行路径的根因诊断的相关研究工作不多,主要是通过建立故障与请求执行路径之间的关联关系,当发现待检测的请求执行路径与已知的存在故障的请求执行路径相匹配时,自动报告故障的根因.

3.3 基于请求执行路径的性能分析

分布式软件系统的性能问题是近年来倍受关注的问题,性能问题往往会影响用户的体验,导致资源浪费与客户流失.但是,由于分布式软件系统往往分布在大量的节点上,组件之间的交互比较复杂,导致对分布式软件系统进行性能分析成为一个难题.传统的性能分析方法通常采用几类资源的监控数据(如 CPU、内存的利用率、响应时间、吞吐量等)检测分布式软件系统的性能问题.但是由于监控数据通常是以单个节点/进程/线程为中心,无法精准地判断涉及大量节点与进程的分布式软件系统的性能问题.

通过将分布式追踪技术产生的请求执行路径与相关的分布式系统的性能指标相结合,如响应延迟、CPU 消耗等,可以在分布式软件系统中进行精准的性能分析[76].基于请求执行路径的性能分析主要需解决两个问题:(1) 请求响应延迟异常检测;(2) 资源消耗异常检测.前者使用响应时间正常的请求作为参照,基于这类请求在正常执行情况下的响应延迟,细粒度地检测请求的响应时间是否出现了不符合预期响应时间的状况,并在请求执行路径中定位引起响应延迟的具体位置;后者结合请求执行路径与分布式软件系统的资源消耗信息(CPU/Memory/Disk 等),构建某类请求在特定资源上的资源消耗模式,当该请求的某类资源消耗与所构建的资源消耗模式不符时,则认为该请求出现了资源消耗异常,并诊断异常资源消耗的原因.

在请求响应延迟异常检测方面,文献[15]提出了一种基于请求执行路径检测请求响应延迟异常的方法.首先,根据请求类型对所有的请求执行路径进行聚类.基于假设:含有请求响应延迟异常的一类请求,其响应延迟的分布会比较分散,文献[15]使用变异系数 CV(coefficient of variation)来衡量每一类请求的响应延迟分布的分散程度,当 CV 大于某阈值时,则认为此类请求中出现了延迟异常.文献[13]使用 KS-检验(Kolmogorov-Smirnov test)检测请求的响应延迟是否符合正常的请求执行的响应延迟的分布.文献[55]基于 Dapper 生成的请求执行路径,根据每次 RPC 调用所耗时间与资源消耗推断本次请求的执行时间,当请求的执行时间与推断的请求执行时间的偏差超过一定阈值时,检测出性能异常并判断性能异常是由请求执行结构变化还是子 RPC 性能异常导致.文献[77]则提出了一种将时间序列化的监控数据与请求的事件序列相关联,从而帮助运维人员诊断性能故障根因的方法.文献[11,27,78,79]同样基于分布式追踪产生的请求执行路径,建立请求正常执行下的请求响应延迟分布的模型,并根据构建的模型进行请求响应延迟异常检测.

在资源消耗异常检测方面,文献[14]将请求执行路径与每个节点上的资源消耗通过时间相关联并进行可视化,辅助运维人员查找故障根因.文献[24]则可以监控每个请求在执行过程中实时的资源消耗,通过可视化发现资源的异常消耗.

在分布式软件系统的性能分析领域,请求执行路径精准刻画分布式软件系统的执行逻辑的特点,使得准确地分析分布式软件系统的性能故障与瓶颈成为可能.对于分布式软件系统,当前的相关研究工作利用请求执行路径作为框架,将请求执行的时间与资源消耗信息嵌入请求执行路径,使用可视化或者统计检验方法来进行分布式软件系统的性能分析,相信未来会有更多研究工作提出结合请求执行路径与时间和资源消耗信息的性能分析方法.

4 分布式追踪技术研究展望

分布式追踪技术是分布式软件系统领域一项重要的技术,不仅在学术界与工业界出现了大量的分布式追踪技术研究工作与分布式追踪系统,也出现了大量的研究工作探索将分布式追踪技术应用到不同的运维场景中,如理解分布式软件系统的架构与行为[65],精准诊断分布式软件系统中的故障[73],检测分布式软件系统中的性能问题[76].通过本文对具体的分布式追踪技术与系统的分析可以看出,已有的分布式追踪技术的研究工作主要集中在以下两个方面:(1) 系统日志及内核事件日志之间的因果关系推断问题,相关研究工作提出不同的推

断方法判断系统日志之间以及内核事件日志之间的因果关系;(2) 基于代码侵入的追踪数据获取问题,相关研究工作主要研究在不同层次软件堆栈中如何设置追踪点并设计请求上下文的传播机制,但在一些问题上的研究仍处于初级阶段,需要研究者进一步的探索与研究,主要包括:(1) 数据读写依赖问题;(2) 通用性问题;(3) 评价问题.因此,本节将从分布式追踪技术中的数据读写依赖问题、通用性问题、评价问题这 3 个方面分析当前分布式追踪技术的不足并探讨未来的研究方向.

4.1 分布式追踪技术中的数据读写依赖问题

随着分布式软件系统的不断发展,为提高分布式软件系统的性能与吞吐量,降低分布式软件系统的耦合度,消息队列尤其是消息中间件在大型分布式软件系统中得到了大量的应用,典型的消息中间件如 RabbitMQ、ActiveMQ、Kafka.在分布式软件系统中,不同的组件将消息发送到消息中间件的消息队列中,其他的组件从消息中间件中接收所订阅的消息并进行处理.消息队列也不仅仅存在于消息中间件中,很多分布式系统自身采用了消息队列来提高性能与吞吐量,如在 Hadoop 的 RPC 通信中,RPC 客户端发送的消息首先会被 RPC 服务端置入消息队列,由 Handler 读取队列中的 RPC 请求并进行真正的方法调用且将调用返回的结果置入另一个消息队列中,由 Responser 将 RPC 调用的结果返回给 RPC 客户端.因此,在分布式软件系统中一个请求的完整处理过程可能会多次经过消息队列.如果分布式追踪技术无法准确捕获对同一消息的读写产生的事件之间的因果关系,则分布式追踪技术将会对一个请求产生多个请求执行路径碎片,从而无法刻画请求在分布式软件系统中完整、准确的处理过程,称其为数据读写依赖问题.随着消息队列在分布式软件系统中越来越广泛的应用,数据读写依赖问题使得分布式追踪技术面临着新的挑战.

数据读写依赖问题已经引起了研究者的关注,Chanda[60]等人提出了一种判断对共享的内存区域进行读写的事件之间因果关系的判断方法,Zhang[80]等人在移动平台中提出了根据对同一事件的读写判定事件之间因果关系判断的方法,Wu[59]等人则基于对遵循一定标准的消息中间件的消息的解析,提取每个消息读取与写入事件中的消息 ID,进而判断同一消息的读写事件的因果关系.但已有的研究工作仅能在特定应用场景中判断特定类型的消息读写之间的因果关系,分布式追踪技术中面临的数据读写依赖问题并没有得到很好的解决.如何处理分布式软件系统中复杂的数据读写依赖问题,进而生成完整、准确的请求执行路径,是分布式追踪技术所面临的一个重要的问题,也将是分布式追踪技术的重要研究方向之一.

4.2 分布式追踪技术的通用性问题

当前,大量的分布式追踪技术仅能追踪特定分布式软件系统中的服务请求[3,81],或者处理基于特定运行时环境[18,19,22,23]、特定系统框架[63,68]及特定企业公共链接库[9,24]的分布式软件系统.而随着开源软件的发展及微服务架构的兴起,分布式软件系统的结构变得更加复杂,一个分布式软件系统可能会由多个不同团队使用不同语言开发的不同的服务与组件组成,导致对一个请求在分布式软件系统的追踪可能需要多个不同分布式追踪技术的共同协作才能完整追踪请求的处理过程[24],但不同分布式追踪技术产生的请求执行路径在粒度、数据结构、因果关系、路径表示方面都有巨大的不同,给分布式追踪技术的应用带来了巨大的挑战.

针对通用型的分布式追踪系统,有的研究者从系统日志入手[8,51],从系统日志中使用统计方法构建请求的执行路径;有的研究者从操作系统层级利用分布式软件系统的系统调用[1]或者产生的内核事件[46]提出分布式追踪技术.从系统日志中推断请求执行路径面临着请求路径的准确性不高的问题,而从操作系统层级构建分布式追踪系统则面临着缺乏上层应用信息、请求执行路径难以理解的问题.如何结合两类提高分布式追踪技术通用性的方法,提出了一种通用的分布式追踪技术将是未来的研究趋势之一.

针对不同分布式追踪技术产生的执行路径的粒度、数据结构、因果关系、路径表示方面的不兼容的问题,Alawneh[82]倡议对其进行标准化,而开源的分布式追踪系统 Zipkin[25]与 Opentracing[26]则基于 Dapper 提出了各自的数据模型,对事件类型、因果关系与请求执行路径提出了规范.但当前的分布式追踪领域仍缺少一种通用的数据模型,能够适用于不同粒度、不通场景下的分布式追踪技术.一个通用的数据模型能够提高不同分布式追踪技术之间的兼容性,且能够加速分布式软件系统中基于请求执行路径的故障诊断、性能分析、系统理解等

各项运维任务的算法的提出,从而发挥分布式追踪技术的价值.

4.3 分布式追踪技术的评价问题

由于当前分布式追踪技术的评价指标不够完善,相关研究工作评价分布式追踪技术主要从该技术所引入的系统性能的开销[9]以及基于请求执行路径的运维任务,如故障诊断[1]的效果等方面来进行,无法完整地评价一个分布式追踪技术,制约了分布式追踪技术的发展.基于对相关研究工作的分析与总结,分布式追踪技术自身的评价指标应该包括以下几个方面.

(1) 开销.分布式追踪技术所引入的开销有 3 个方面:① 将分布式追踪技术应用于分布式软件系统后,分布式软件系统中请求响应时间的增加;② 将分布式追踪技术应用于分布式软件系统后,分布式软件系统的吞吐量(通常是指分布式软件系统所能并发处理的请求的数量)的降低;③ 为实现分布式追踪,所需要的修改分布式软件系统的代码的工作量.

(2) 通用性.分布式追踪系统的通用性代表了分布式追踪技术满足不同类型、不同架构、不同语言的分布式软件系统的追踪要求的能力.

(3) 准确性.准确性是指分布式追踪技术产生的请求路径中事件之间的正确因果关系占所有捕获的事件之间因果关系的百分比.

(4) 完整性.完整性描述了分布式追踪技术产生的请求执行路径完整刻画请求的端到端处理过程的能力.

相关的研究工作对如何评价请求响应时间开销[19]、系统吞吐量开销[9]以及代码修改的开销[20]提供了可量化的评价方法,但如何对分布式追踪技术的通用性、准确性与完整性进行评价则缺少相关的研究工作与标准.文献[31]虽然对其分布式追踪技术的准确性与完整性进行了评价,但其评价方法依赖于专家所进行的人工判断,无法应用到其他分布式追踪技术的评价中.因此,一个用于评价分布式追踪技术的基准(benchmark)系统以及在相应基准系统上对分布式追踪技术的开销、通用性、准确性及完整性进行评价的指标体系对于促进分布式追踪技术的发展是必要的.相应的研究工作也会对分布式追踪技术的完善与应用产生积极的影响.

5 结束语

分布式追踪技术是精准刻画分布式软件系统的行为与状态的重要手段,对满足分布式软件系统中的故障检测、性能调优、系统理解、资源审计等一系列运维任务有着重要的意义,其研究受到了工业界和学术界的广泛关注.但当前分布式追踪技术仍面临着新的挑战,如无法准确地判断因数据读写依赖导致的事件之间的因果关系,分布式追踪技术的通用性差,缺乏有效的对分布式追踪技术的评价指标等,这些都需要进一步的研究.

本文从分布式追踪技术的基本概念出发,提出了分布式追踪技术的研究框架,并在该框架下详细地分析并总结了已有的分布式追踪技术的研究工作.通过整理总结已有的分布式追踪技术及其应用的相关工作,进一步分析了分布式追踪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提供了一些思路.

致谢 在此,我们向对本文工作给予支持和建议的同行表示衷心的感谢.

References:

[1] Yong Y, Long W, Jing G, Ying L. Transparently capturing execution path of service/job request processing. In: Proc. of the Int'l Conf. on Service-oriented Computing. Springer-Verlag, 2018. 879–887.

[2] Zhao X, Kirk R, Yu L, *et al*. Log20: Fully automated optimal placement of log printing statements under specified overhead threshold. In: Proc. of the 26th Symp. on Operating Systems Principles. ACM, 2017. 565–581.

[3] Thereska E, Salmon B, Strunk J, Wachs M, Abd-El-Malek M, Lopez J, Ganger GR. Stardust: Tracking activity in a distributed storage system. ACM SIGMETRICS Performance Evaluation Review, 2006,34(1):3–14.

[4] Cantrill MB, Michael WS, Adam HL. Dynamic instrumentation of production systems. In: Proc. of the USENIX Annual Technical Conf. USENIX. 2004. 15–28.

[5] Gupta M, Anindya N, Manoj KA, Gautam K. Discovering dynamic dependencies in enterprise environments for problem determination. In: Proc. of the Int'l Workshop on Distributed Systems: Operations and Management. Springer-Verlag, 2003. 221–233.

[6] Jonathan, M. End-to-end tracing: Adoption and use cases, survey. Brown University, 2017. http://cs.brown.edu/people/jcmace/papers/mace2017survey.pdf

[7] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978,21(7):558–565.

[8] 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.

[9] Sigelman BH, Barroso LA, Burrows M, Mike B, Pat S, Manoj P, Donald B, Saul J, Chandan S. Dapper, a large-scale distributed systems tracing infrastructure. Google Technical Report, Google Inv., 2010.

[10] Kitajima S, Matsuoka N. Inferring calling relationship based on external observation for microservice architecture. In: Proc. of the Int'l Conf. on Service-oriented Computing. Springer-Verlag, 2017. 229–237.

[11] Zhang Z, Zhan J, Li Y, Lei W, Dan M, Bo S. Precise request tracing and performance debugging for multi-tier services of black boxes. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems & Networks. IEEE, 2009. 337–346.

[12] Bahl P, Chandra R, Greenberg A, Srikanth K, David AM, Ming Z. Towards highly reliable enterprise network services via inference of multi-level dependencies. ACM SIGCOMM Computer Communication Review, 2007,37(4):13–24.

[13] Sambasivan RR, Zheng AX, De RM, Elie K, Spencer W, Mchael S, William W, Lianghong X, Grrgory RG. Diagnosing performance changes by comparing request flows. In: Proc. of the Symp. on Networked Systems Design and Implementation (NSDI). USENIX, 2011,5:1.1–5.8.

[14] Lai CA, Kimball J, Zhu T, Qingyang Q, Claton P. MilliScope: A fine-grained monitoring framework for performance debugging of n-tier Web services. In: Proc. of the 37th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS). IEEE, 2017. 92–102.

[15] Mi H, Wang H, Zhou Y, Micharl RTL, Hua C. Toward fine-grained, unsupervised, scalable performance diagnosis for production cloud computing systems. IEEE Trans. on Parallel and Distributed Systems, 2013,24(6):1245–1255.

[16] Reynolds P, Killian CE, Wiener JL, Jeffrey CM, Mehul AS, Amin V. Pip: Detecting the unexpected in distributed systems. In: Proc. of the Symp. on Networked Systems Design and Implementation (NSDI). USENIX, 2006,6:9–9.

[17] Chen M, Emre K, Anthony A, Armondo F, Eric B. Using runtime paths for macroanalysis. In: Proc. of the USENIX Workshop on Hot Topics in Operating Systems (HotOS). USENIX, 2003. 79–84.

[18] Chen MY, Kiciman E, Fratkin E, Armondo F, Eric B. Pinpoint: Problem determination in large, dynamic internet services. In: Proc. Int'l Conf. on Dependable Systems and Networks. IEEE, 2002. 595–604.

[19] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. ACM Trans. on Computer Systems (TOCS), 2018,35(4):11.

[20] Mace J, Roelke R, Fonseca R. Pivot tracing: Dynamic causal monitoring for distributed systems. In: Proc. of the 25th Symp. on Operating Systems Principles (SOSP). ACM, 2015. 378–393.

[21] Barham P, Donnelly A, Isaacs R, Richad. Using magpie for request extraction and workload modelling. In: Proc. of the Symp. on Operating Systems Principles (SOSP). USENIX, 2004,4:18–18

[22] Barham P, Rebecca I, Richard M, Dushyanth N. Magpie: Online modelling and performance-aware systems. In: Proc. of the USENIX Workshop on Hot Topics in Operating Systems (HotOS). USENIX, 2003. 85–90.

[23] Li D, Mickens J, Nath S, Lenin R. Domino: Understanding wide-area, asynchronous event causality in Web applications. In: Proc. of the 6th ACM Symp. on Cloud Computing. ACM, 2015. 182–188.

[24] Kaldor J, Mace J, Bejda M, Edison G, Wiktor K, Joe O, Kian WO, Bill S, Pingjia S, Brendan V, Vinod V, Kaushik V, Yee JS. Canopy: An end-to-end performance tracing and analysis system. In: Proc. of the 26th Symp. on Operating Systems Principles (SOSP). ACM, 2017. 34–50.

[25] https://zipkin.io/

[26] https://opentracing.io/

[27] Fonseca R, Freedman MJ, Porter G. Experiences with tracing causality in networked services. In: Proc. of the Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN). USENIX, 2010,10:10–10.

[28] Fonseca R, Porter G, Katz R H, Scott S, Ion S. X-trace: A pervasive network tracing framework. In: Proc. of the 4th USENIX Conf. on Networked systems design & implementation (NSDI). USENIX Association, 2007. 20.

[29] Attariyan M, Chow M, Flinn J. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In: Proc. of the 10th Symp. on Operating Systems Design and Implementation (OSDI). USENIX, 2012. 307–320.

[30] Pham C, Wang L, Tak BC, Salman B, Chunqiang T, Zbigniew K, Ravishankar KI. Failure diagnosis for distributed systems using targeted fault injection. IEEE Trans. on Parallel and Distributed Systems, 2017,28(2):503–516.

[31] Tak BC, Tang C, Zhang C, Sriram G, Bhuvan U, Rong NC. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In: Proc. of the USENIX Annual technical conference (ATC). USENIX, 2009.

[32] Koskinen E, Jannotti J. Borderpatrol: Isolating events for black-box tracing. ACM SIGOPS Operating Systems Review, 2008,42(4): 191–203.

[33] Reynolds P, Wiener JL, Mogul JC, Marcos KA, Amin V. WAP5: Black-box performance debugging for wide-area systems. In: Proc. of the 15th Int'l Conf. on World Wide Web. ACM, 2006. 347–356.

[34] Neves F, Machado N, Jose P. Falcon: A practical log-based analysis tool for distributed systems. In: Proc. of the 48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN). IEEE, 2018. 534–541.

[35] Aguilera MK, Mogul JC, Wiener JL, Patrick R, Athicha M. Performance debugging for distributed systems of black boxes. ACM SIGOPS Operating Systems Review, 2003,37(5):74–89.

[36] Du M, Feifei L, Guineng Z, Vivek S. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In: Proc. of the 2017 ACM SIGSAC Conf. on Computer and Communications Security. ACM, 2017. 1285–1298.

[37] Zhao X, Rodrigues K, Luo Y, Ding Y, Michael S. Non-intrusive performance profiling for entire software stacks based on the flow reconstruction principle. In: Proc. of the Symp. on Operating Systems Design and Implementation (OSDI). 2016. 603–618.

[38] Tak BC, Tao S, Yang L, Chao Z, Yaoping R. LOGAN: Problem diagnosis in the cloud using log-based reference models. In: Proc. of the 2016 IEEE Int'l Conf. on Cloud Engineering (IC2E). IEEE, 2016. 62–67.

[39] Abrahamson J, Beschastnikh I, Brun Y, Michael DE. Shedding light on distributed system executions. In: Companion Proc. of the 36th Int'l Conf. on Software Engineering. ACM, 2014. 598–599.

[40] Zhao X, Zhang Y, Lion D, Muhammud F, Yu L, Ding Y, Micheal S. Lprof: A non-intrusive request flow profiler for distributed systems. In: Proc. of the Symp. on Operating Systems Design and Implementation (OSDI). USENIX, 2014. 629–644.

[41] Ivan B, Yuriy B, Micheal DE, Arvind K. Inferring models of concurrent systems from logs of their behavior with CSight. In: Proc. of the 36th Int'l Conf. on Software Engineering (ICSE). ACM, 2014. 468–479.

[42] Tan J, Kavulya S, Gandhi R, Priya N. Visual, log-based causal tracing for performance debugging of mapreduce systems. In: Proc. of the 30th IEEE Int'l Conf. on Distributed Computing Systems. IEEE, 2010. 795–806.

[43] Fu Q, Lou JG, Wang Y, Jiang L. Execution anomaly detection in distributed systems through unstructured log analysis. In: Proc. of the 9th IEEE Int'l Conf. on Data Mining. IEEE, 2009. 149–158.

[44] Anandkumar A, Bisdikian C, Agrawal D. Tracking in a Spaghetti Bowl: Monitoring transactions using footprints. ACM SIGMETRICS Performance Evaluation Review, 2008,36(1):133–144.

[45] Xu H, Ning X, Zhang H, Junghwan R, Guofei J. Pinfer: Learning to infer concurrent request paths from system kernel events. In: Proc. of the IEEE Int'l Conf. on Autonomic Computing (ICAC). IEEE, 2016. 199–208.

[46] Zhang H, Rhee J, Arora N, Sanhan G, Guofei J, Kenji Y, Dongyan X. CLUE: System trace analytics for cloud service performance diagnosis. In: Proc. of the IEEE Network Operations and Management Symp. (NOMS). IEEE, 2014. 1–9.

[47] Erlingsson Ú, Peinado M, Peter S, Mihai B, Gloria MR. Fay: Extensible distributed tracing from kernels to clusters. ACM Trans. on Computer Systems (TOCS), 2012,30(4):13.

[48] Mace, J, Peter B, Rodrigo F, Madanlal M. Retro: Targeted resource management in multi-tenant distributed systems. In: Proc. of the 12th USENIX Symp. on Networked Systems Design and Implementation (NSDI). USENIX, 2015. 589–603.

[49] Gschwind T, Eshghi K, Garg PK, Klaus W. 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.

[50] Barham P, Donnelly A, Isaacs R, Richard M. Using magpie for request extraction and workload modelling. In: Proc. of the Symp. on Operating Systems Design and Implementation (OSDI). USENIX, 2004,4:18–18.

[51] Yu X, Joshi P, Xu J, Guoliang J. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. ACM SIGPLAN Notices, 2016,51(4):489–502.

[52] Tan J, Pan X, Kavulya S, Gandhi R, Narasimhan P. SALSA: Analyzing logs as state machines. In: Proc. of the USENIX Workshop on the Analysis of System Logs (WASL). USENIX, 2008. 6.

[53] Wang T, Perng C, Tao T, *et al*. A temporal data-mining approach for discovering end-to-end transaction flows. In: Proc. of the IEEE Int'l Conf. on Web Services. IEEE, 2008. 37–44.

[54] Mi HB, Wang HM, Cai H, *et al*. P-Tracer: Path-based performance profiling in cloud computing systems. In: Proc. of the 36th IEEE Annual Computer Software and Applications Conf. IEEE, 2012. 509–514.

[55] Ostrowski K, Mann G, Sandler M. Diagnosing latency in multi-tier black-box services. In: Proc. of the 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS). ACM, 2011.

[56] Kc K, Gu XH. ELT: Efficient log-based troubleshooting system for cloud computing infrastructures. In: Proc. of the 30th IEEE Int'l Symp. on Reliable Distributed Systems. IEEE, 2011. 11–20.

[57] Tak BC, Tang C, Zhang C, Sriram G, Bhuvan U, Rong NC. vPath: Precise discovery of request processing paths from black-box observations of thread and network activities. In: Proc. of the USENIX Annual Technical Conf (ATC). USENIX, 2009.

[58] Cai H, Douglas T. Distea: Efficient dynamic impact analysis for distributed systems. arXiv Preprint, arXiv:1604.0463, 2016.

[59] Wu LJ, Li HW, Cheng YJ, *et al*. Application dependency tracing for message-oriented middleware. In: Proc. of the 16th Asia-Pacific Network Operations and Management Symp. IEEE, 2014. 1–6.

[60] Chanda A, Cox AL, Zwaenepoel W. Whodunit: Transactional profiling for multi-tier applications. ACM SIGOPS Operating Systems Review, 2007,41(3):17–30.

[61] Kobayashi S, Kensuke F, Hiroshi E. Mining causes of network events in log data with causal inference. In: Proc. of the IFIP/IEEE Symp. on Integrated Network and Service Management (IM). IEEE, 2017. 45–53.

[62] Kanuparthy P, Dai Y, Pathak S, Sambit S, Theophilus B, Mojgan G, Narayan PPS. YTrace: End-to-end performance diagnosis in large cloud and content providers. arXiv Preprint, arXiv:1602.03273, 2016.

[63] Mann G, Sandler M, Krushevskaja D, Sudipto G, Eyar E. Modeling the parallel execution of black-box services. In: Proc. of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud). USENIX, 2011.

[64] Guo Z, Zhou D, Lin HX, *et al*. G2: A graph processing system for diagnosing distributed systems. In: Proc. of the USENIX Annual Technical Conf (ATC). USENIX, 2011.

[65] Gu J, Wang L, Yang Y, *et al*. KEREP: Experience in extracting knowledge on distributed system behavior through request execution path. In: Proc. of the IEEE Int'l Symp. on Software Reliability Engineering Workshops (ISSREW). IEEE, 2018. 30–35.

[66] Israr T, Murray W, Greg F. Interaction tree algorithms to extract effective architecture and layered performance models from traces. Journal of Systems and Software, 2007,80(4):474–492.

[67] Abdelwahab HL, Timothy L. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: Proc. of the 14th IEEE Int'l Conf. on Program Comprehension (ICPC). IEEE, 2006. 181–190.

[68] Moc J, David AC. Understanding distributed systems via execution trace data. In: Proc. of the 9th Int'l Workshop on Program Comprehension (IWPC). IEEE, 2001. 60–67.

[69] Kuhlenkamp J, Markus K. Costradamus: A cost-tracing system for cloud-based software services. In: Proc. of the Int'l Conf. on Service-oriented Computing. Springer-Verlag, 2017. 657–672.

[70] Fonseca R, Dutta P, Phillip L, Ion S. Quanto: Tracking energy in networked embedded systems. In: Proc. of the Symp. on Operating Systems Design and Implementation (OSDI). USENIX, 2008,8:323–338.

[71] Enck W, Gilbert P, Han S, Vasant T, Byung-gon C, Landon PC, Jaeyeon J, Patrick M, Anmol NS. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans. on Computer Systems (TOCS), 2014,32(2):5.

[72] Sambasivan RR, Shafer I, Mazurek ML, Gregory RG. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. IEEE Trans. on Visualization and Computer Graphics, 2013,19(12):2466–2475.

[73] Chen MY, Accardi A, Kiciman E, Jim L, Dave P, Armondo F, Eric B. Path-based faliure and evolution management. In: Proc. of the 1st Conf. on Symp. on Networked Systems Design and Implementation. USENIX Association, 2004. 23.

[74] Kavulya SP, Daniels S, Joshi K, Matti H, Rajeev G, Priya N. Draco: Statistical diagnosis of chronic problems in large distributed systems. In: Proc. of the IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN 2012). IEEE, 2012. 1–12.

[75] Yuan D, Mai HH, Xiong WW, *et al*. SherLog: Error diagnosis by connecting clues from run-time logs. ACM SIGARCH Computer Architecture News, 2010, 143–154.

[76] Wang C, Kavulya SP, Tan J, Liting H, Mahendra K, Mike K, Karsten S, Priya N, Rajeev G. Performance troubleshooting in data centers: An annotated bibliography. ACM SIGOPS Operating Systems Review, 2013,47(3):50–62.

[77] Luo C, Lou JG, Lin Q, Qiang F, Rui D, Dongmei Z, Zhe W. Correlating events with time series for incident diagnosis. In: Proc. of the 20th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining. ACM, 2014. 1583–1592.

[78] Chen P, Qi Y, Hou D. CauseInfer: Automated end-to-end performance diagnosis with hierarchical causality graph in cloud environment. IEEE Trans. on Services Computing, 2016,12(2):214–230.

[79] Chen P, Qi Y, Zheng P, Di H. Causeinfer: Automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems. In: Proc. of the IEEE INFOCOM & IEEE Conf. on Computer Communications. IEEE, 2014. 1887–1895.

[80] Zhang L, Bild DR, Dick RP, Mao ZM, Peter D. Panappticon: Event-based tracing to measure mobile application and platform performance. In: Proc. of the Int'l Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS). IEEE, 2013. 1–10.

[81] http://incubator.apache.org/projects/htrace.html

[82] Alawneh L, Hamou-Lhadj A. Execution traces: A new domain that requires the creation of a standard metamodel. In: Proc. of the Int'l Conf. on Advanced Software Engineering and Its Applications. Springer-Verlag, 2009. 253–263.

杨勇(1993－),男,博士生,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.

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.