软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2023,34(12):5578−5596 [doi: 10.13328/j.cnki.jos.006755]

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

共用数据导向的分布式系统失效恢复缺陷检测*

高 钰[1,2], 王 栋[1,2], 戴千旺[1,2], 窦文生[1,2,3,4], 魏 峻[1,2,3,4]

[1](中国科学院 软件研究所, 北京 100190)

[2](中国科学院大学, 北京 100049)

[3](中科南京软件技术研究院, 江苏 南京 210000)

[4](中国科学院大学南京学院, 江苏 南京 211135)

通信作者: 窦文生, E-mail: wsdou@otcaix.iscas.ac.cn

摘 要: 分布式系统的可靠性和可用性至关重要. 然而, 不正确的失效恢复机制及其实现会引发失效恢复缺陷, 威胁分布式系统的可靠性和可用性. 只有发生在特定时机的节点失效才会触发失效恢复缺陷, 因此, 检测分布式系统中的失效恢复缺陷具有挑战性. 提出了一种新方法 Deminer 来自动检测分布式系统中的失效恢复缺陷. 在大规模分布式系统中观察到, 同一份数据 (即共用数据) 可能被一组 I/O 写操作存储到不同位置 (如不同的存储路径或节点). 而打断这样一组共用数据写操作执行的节点失效更容易触发失效恢复缺陷. 因此, Deminer 以共用数据的使用为指导, 通过自动识别和注入这类容易引发故障的节点失效来检测失效恢复缺陷. 首先, Deminer 追踪目标系统的一次正确执行中关键数据的使用. 然后, Deminer 基于执行轨迹识别使用共用数据的 I/O 写操作对, 并预测容易引发错误的节点失效注入点. 最后, Deminer 通过测试预测的节点失效注入点以及检查故障征兆来暴露和确认失效恢复缺陷. 实现了 Deminer 原型工具, 并在 4 个流行的开源分布式系统 ZooKeeper、HBase、YARN 和 HDFS 的最新版本上进行了验证. 实验结果表明 Deminer 方法能够有效检测分布式系统中的失效恢复缺陷. Deminer 已经检测到 6 个失效恢复缺陷.

关键词: 失效恢复缺陷; 缺陷检测; 故障注入; 失效恢复; 分布式系统

中图法分类号: TP311

中文引用格式: 高钰, 王栋, 戴千旺, 窦文生, 魏峻. 共用数据导向的分布式系统失效恢复缺陷检测. 软件学报, 2023, 34(12): 5578–5596. http://www.jos.org.cn/1000-9825/6755.htm

英文引用格式: Gao Y, Wang D, Dai QW, Dou WS, Wei J. Common Data Guided Crash Recovery Bug Detection for Distributed Systems. Ruan Jian Xue Bao/Journal of Software, 2023, 34(12): 5578–5596 (in Chinese). http://www.jos.org.cn/1000-9825/6755.htm

Common Data Guided Crash Recovery Bug Detection for Distributed Systems

GAO Yu[1,2], WANG Dong[1,2], DAI Qian-Wang[1,2], DOU Wen-Sheng[1,2,3,4], WEI Jun[1,2,3,4]

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

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

[3](Nanjing Institute of Software Technology, Nanjing 210000, China)

[4](University of Chinese Academy of Sciences, Nanjing, Nanjing 211135, China)

Abstract: The critical reliability and availability of distributed systems are threatened by crash recovery bugs caused by incorrect crash recovery mechanisms and their implementations. The detection of crash recovery bugs, however, can be extremely challenging since these bugs only manifest themselves when a node crashes under special timing conditions. This study presents a novel approach Deminer to automatically detect crash recovery bugs in distributed systems. Observations in the large-scale distributed systems show that node crashes

* 基金项目: 国家自然科学基金 (62072444, 61732019); 中国科学院前沿科学重点研究项目 (QYZDJ-SSW-JSC036); 中国科学院青年创新促进会 (2018142)
收稿时间: 2022-01-07; 修改时间: 2022-04-21; 采用时间: 2022-07-25; jos 在线出版时间: 2022-10-27
CNKI 网络首发时间: 2023-03-03

that interrupt the execution of related I/O write operations, which store a piece of data (i.e., common data) in different places, e.g., different storage paths or nodes, are more likely to trigger crash recovery bugs. Therefore, Deminer detects crash recovery bugs by automatically identifying and injecting such error-prone node crashes under the usage guidance of common data. Deminer first tracks the usage of critical data in a correct run. Then, it identifies I/O write operation pairs that use the common data and predicts error-prone injection points of a node crash on the basis of the execution trace. Finally, Deminer tests the predicted injection points of the node crash and checks failure symptoms to expose and confirm crash recovery bugs. A prototype of Deminer is implemented and evaluated on the latest versions of four widely used distributed systems, i.e., ZooKeeper, HBase, YARN, and HDFS. The experimental results show that Deminer is effective in finding crash recovery bugs. Deminer has detected six crash recovery bugs.

Key words: crash recovery bug; bug detection; fault injection; crash recovery; distributed system

随着越来越多的数据与计算从本地往云端迁移, 大规模分布式系统已经在大型互联网, 如阿里巴巴、百度、腾讯、谷歌等得到广泛部署应用, 如存储系统[1−3]、集群管理服务[4,5]、计算框架[6]以及同步服务[7]等, 为海量规模的用户提供可靠支撑. 然而, 分布式系统中的节点不可避免地会发生失效 (crash)[8], 导致不一致的系统状态, 影响分布式系统的可用性和可靠性. 虽然分布式系统引入了各种各样的失效恢复机制来容忍节点失效, 对于开发者而言, 设计并实现没有缺陷的失效恢复协议, 保证集群能够正确地从所有的节点失效场景中恢复, 是一个巨大的挑战. 不正确的失效恢复机制及其实现会引入错综复杂的失效恢复缺陷[9].

在所有的失效场景中, 导致不一致系统状态的节点失效更易引发失效恢复缺陷. 我们观察到, 在分布式系统中, 一个节点会把同一份数据通过多个 I/O 写操作存储在节点外多个位置 (也就是多个 I/O 写操作使用共用数据), 例如集群中其他节点或一个存储系统如本地文件系统或分布式文件系统 HDFS 中. 一个节点失效会导致该节点内所有的内存状态立刻丢失, 而被这个节点存储在节点外的数据则依然可以被访问并影响系统后续的行为. 如果一个节点失效打断一对共用数据写操作的执行, 则会导致不一致的系统状态, 进一步可能使后续的恢复行为失败.

图 1 展示了一个当节点失效发生在使用共用数据的两个 I/O 写操作之间时, 正确的失效恢复行为. 如图 1 所示, 节点 2 和分布式文件系统都存储着数据 d 的值 d_{old} (第 1 步). 数据 d 的值发生变化时, 节点 1 首先将新的值 d_{new} 更新到分布式文件系统中 (第 2 步). 然后节点 1 在将 d_{new} 更新到节点 2 之前发生失效 (第 3 步). 此时, 节点 1 所有的内存数据立刻丢失, 而存储在分布式文件系统中的 d_{new} 值和存储在节点 2 上的 d_{old} 值仍然可以被集群所访问. 在一个正确的恢复行为中, 分布式集群会从这个不一致的系统状态中恢复, 最后节点 2 和分布式文件系统对数据 d 的值会达成一致, 例如 d_{cons} (第 4 步). 在这里, 一致的值 d_{cons} 可能是旧的值 d_{old} 、新的值 d_{new} , 或者一个其他的值. 当集群无法正确地从这个不一致的系统状态中恢复时, 就会触发一个失效恢复缺陷.

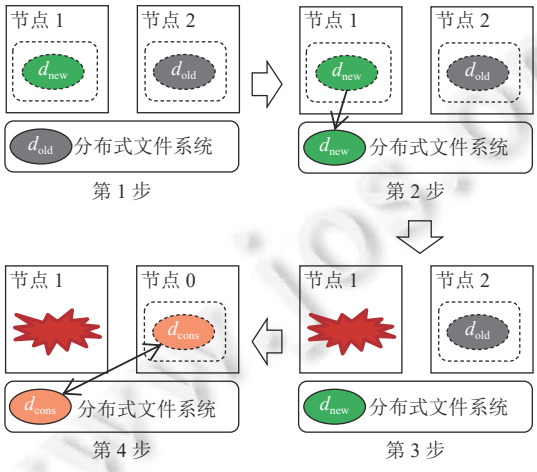


图 1 一个正确的失效恢复例子

失效恢复缺陷难以避免. 在分布式系统中, 用“锁”来保护一组操作的执行不被节点失效所打断非常困难. 同时, 对于开发者而言, 预见所有可能导致不一致系统状态的节点失效场景具有挑战. 另外, 由于导致不一致系统状态的

节点失效必须发生在特定的时机, 且使用共用数据的 I/O 写操作可能具有非常小的缺陷触发时间窗口, 因此在内部测试 (in-house testing) 中提早暴露失效恢复缺陷也十分困难. 此外, 失效恢复缺陷难以检测和诊断. 例如, 图 1 所示的失效场景中, 识别出向节点外写出同一份数据的共用数据写操作具有挑战. 这些写操作使用的共用数据可能与使用该数据的 I/O 写操作位于同一节点, 也可能源自集群中另外一个节点.

最近已经有一些研究可以用于检测与节点失效相关的缺陷[10]. 其中, 随机故障注入框架[11,12]随机地向目标系统中注入节点失效, 试图通过多次尝试来暴露具有较小触发时间窗口的缺陷. 分布式系统模型检查器[12−17]通过系统地探索包括节点失效在内的非确定性分布式事件的所有可能执行序列来进行缺陷检测, 但是分布式系统模型检查器会有状态空间爆炸问题. 基于程序分析的缺陷检测工具 FCatch[18]和 CrashTuner[19]则用于分别检测与故障时机相关的缺陷以及与元信息相关的缺陷.

本文中, 我们提出了 Deminer, 一种新的分布式系统失效恢复缺陷检测方法. Deminer 通过在使用共用数据的 I/O 写操作之间注入节点失效和节点重启来检测失效恢复缺陷. 我们通过分析真实的分布式系统以及真实的失效恢复缺陷, 总结了 3 种缺陷模式. 基于缺陷模式, Deminer 首先通过观察目标系统的一次正确执行来追踪目标系统中的数据使用, 然后根据执行轨迹识别使用共用数据的相关 I/O 写操作对, 并预测会导致不一致系统状态的、容易引发错误的节点失效注入点. 最后, Deminer 测试每一个预测的节点失效注入点, 注入相应的节点失效事件及节点重启事件, 并通过检查故障征兆识别失效恢复缺陷.

我们已经实现了 Deminer 工具原型, 并在 4 个流行的开源分布式系统 ZooKeeper[20]、HBase[21]、YARN[22]和 HDFS[23]的最新发布版本上进行了验证. Deminer 已经成功检测到 6 个失效恢复缺陷. 这些失效恢复缺陷会导致集群无服务、节点宕机、操作失败、数据过时以及误导性错误信息. 在这些缺陷中, 1 个缺陷是已知缺陷, 但其修复没有被合并到我们所测试的最新系统版本上, 而其他缺陷是未知缺陷.

本文第 1 节通过一个真实的失效恢复缺陷来阐述本文的研究动机, 并介绍了我们的缺陷模式和面临的挑战. 第 2 节介绍本文的缺陷检测方法. 第 3 节介绍了方法的具体实现. 第 4 节通过实验验证了所提方法的有效性. 第 5 节介绍了所提方法的局限性. 第 6 节介绍了分布式系统故障注入及缺陷检测研究现状. 第 7 节总结全文.

1 研究动机

在本节中, 我们首先通过一个真实的失效恢复缺陷引入我们的研究动机, 然后介绍我们的缺陷模型, 最后介绍我们的缺陷检测方法所面临的主要技术挑战.

1.1 失效恢复缺陷实例

在分布式系统中, 使用同一份数据的 I/O 写操作通常被期待一起执行. 打断这些操作执行的节点失效会导致不一致的系统状态. 如果不一致的系统状态无法被后续的恢复过程正确处理, 就会引发失效恢复缺陷. 如图 2 所示, 展示了一个 ZooKeeper 当中的真实失效恢复缺陷示例.

当一个 leader 节点开始工作时, 它首先会启动一个线程来等待新的 follower 节点的连接请求 (第 4 行). 随后 leader 节点提出了当前的 epoch 值, 并在收到集群中大多数 follower 节点的响应后, 把新的 epoch 值保存在 epoch 域变量 (第 1 行) 中. 随后, leader 节点用新的 epoch 值设置当前事务 ID (即 zxid, 第 6 行), 并且用新的 zxid 更新 ZooKeeper 数据库 (第 7 行).

此时, 一个 ZooKeeper 节点被启动并决定跟随当前 leader 节点, 并与 leader 节点同步 (第 17–20 行). 该 follower 节点首先通过 LEADERINFO 消息 (①→②) 从 leader 节点收到最新的事务 ID. 然后 follower 节点从另一个消息 snapshot 消息 (③→④) 中反序列化快照数据用于数据同步. LEADERINFO 消息和 snapshot 消息都传输了同一份数据, 即新的 epoch 值. 随后, follower 节点首先把当前内存数据状态保存在 snapshot 文件中 (⑤), 然后将新的 epoch 值保存在 currentEpoch 文件中 (⑥). 在这个例子中, I/O 写操作⑤和⑥使用共用数据, 即源自 leader 节点的新的 epoch 值.

当 follower 节点在⑤和⑥之间失效时, snapshot 文件将保存最新的 epoch 值, 而 currentEpoch 文件仍然保存着

旧的 epoch 值. 这两个文件中不一致的 epoch 值将导致该 follower 节点无法在没有人工干预的情况下重启. 如果 follower 节点在更新 snapshot 文件前失效, 或在更新 currentEpoch 文件后失效, 则 follower 节点不会从 snapshot 文件中读到一个更大的 epoch 值, 因此能够成功重启.

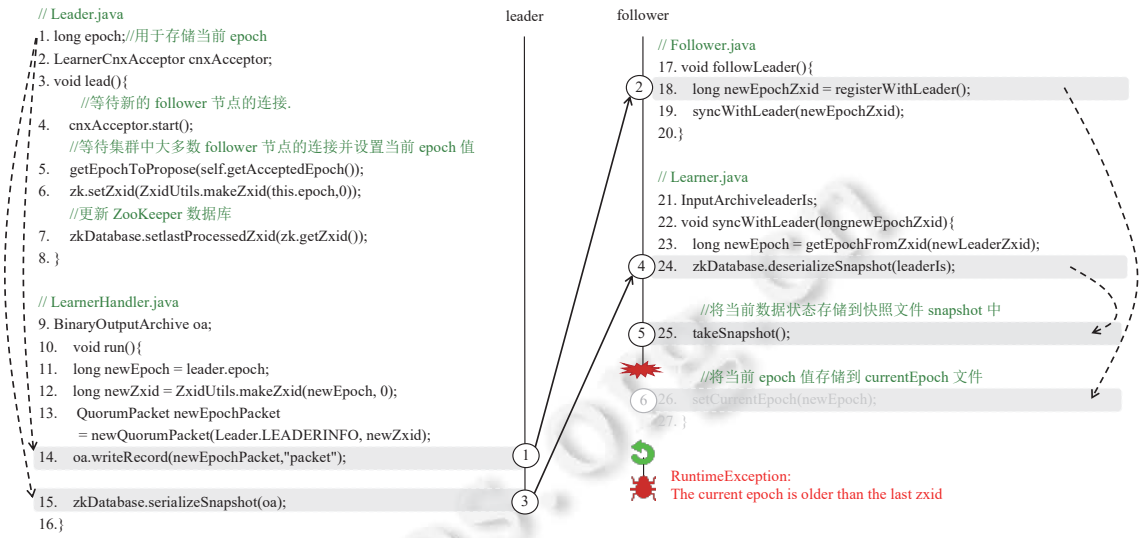


图 2 一个 ZooKeeper 中的失效恢复缺陷示例 (实线箭头表示消息; 虚线箭头表示节点内数据流)

1.2 缺陷模式抽象

要触发我们在第 1.1 节描述的失效恢复缺陷, Deminer 需要在使用同一份数据的 I/O 写操作之间注入节点失效及相应的节点重启. 下面, 我们将详细介绍我们分析总结的缺陷模式. 该缺陷模式指导了 Deminer 方法的设计.

相关术语如下: 本文中, I/O 写操作指可以把数据存储在节点外的存储系统更新操作或节点间的消息发送操作. 在本文中, 我们考虑 3 种存储系统: 本地文件系统、分布式文件系统 HDFS 和分布式 key-value 存储系统 ZooKeeper. 由同一个节点所执行的多个 I/O 写操作可能使用同一份数据 (我们称为共用数据), 并把该数据存储在节点外多个位置 (我们称为目标路径), 比如不同的节点或存储路径. 我们把这些使用共用数据的 I/O 写操作称为共用数据写操作. 由于我们主要考虑分布式系统中不同目标路径下数据的不一致, 而不是考虑传统文件系统中关注的打断对同一个目标文件的更新所引发的不一致, 因此我们只关注具有不同目标路径的共用数据写操作.

如图 3 所示, 在分析了真实的分布式系统以及失效恢复缺陷后, 我们总结了 3 种缺陷模式, 刻画了实际分布式场景中的共用数据写操作对, 以及可以打断共用数据写操作对执行的、导致不一致系统状态的节点失效点. 在这 3 种缺陷模式中, 一个 I/O 写操作使用的数据指可以通过该 I/O 写操作传播到节点外的数据. 一个 I/O 写操作所使用的数据可能最初由当前节点获得, 也可能通过集群中其他节点传播而来. 如果一对 I/O 写操作分别使用数据 $Data_1$ 和 $Data_2$, 则它们共同使用的数据 d 可以通过计算数据 $Data_1$ 和 $Data_2$ 的交集 (即 $common(Data_1, Data_2)$) 得到. 如果数据 d 不为空, 且两个 I/O 写操作的目标路径不同, 则这两个 I/O 写操作是一对共用数据 I/O 写操作. 发生在一对共用数据 I/O 写操作之间的节点失效会使集群中不同位置对数据 d 的认知不同. 这种不一致性蕴含着失效恢复缺陷. 具体而言, 我们的缺陷模式如下所述.

● 单节点共用数据写操作相关失效恢复缺陷. 如图 3(a) 所示, 一个节点 $Node_1$ 通过两个存储系统更新操作 $Write_{path_1}(Data_1)$ 和 $Write_{path_2}(Data_2)$, 将共用数据 d 分别写入两个不同的存储路径 $path_1$ 和 $path_2$ 中. 这两个存储系统更新操作是一对共用数据写操作. 使 $Node_1$ 在 $Write_{path_1}$ 和 $Write_{path_2}$ 之间失效 ($Crash$), 会使 $Write_{path_2}$ 无法被执行, 从而使路径 $path_1$ 和 $path_2$ 下的数据内容不一致. 这个不一致的状态会被集群中的其他节点 (对分布式存储系统而言) 或重启的 $Node_1$ 节点访问, 容易使后续恢复过程出错, 引发失效恢复缺陷. 在第 1.1 节描述的失效恢复缺陷

就属于这种缺陷模式.

特别地, 存储系统更新操作 $Write_{path_1}$ 和 $Write_{path_2}$ 可能是一个路径/文件创建操作、一个路径/文件删除操作或对路径/文件的写操作. 在分布式系统中, 把重要信息作为路径字符串的一部分保存是一种常见操作. 例如, 在 HBase 中, 当创建一个新的表 mytable 时, HBase 会用该表名在 ZooKeeper 上创建一个新的 ZooKeeper 节点 (znode):"/hbase/table/mytable". 因此, 对于一个路径/文件创建及删除操作, 其所使用的数据 (如 $Data_1$ 和 $Data_2$) 包含路径字符串数据 (如 $path_1$ 和 $path_2$) 以及写入该路径/文件的数据. 而对于文件写操作, 其所使用的数据仅包含存储在路径/文件中的数据.

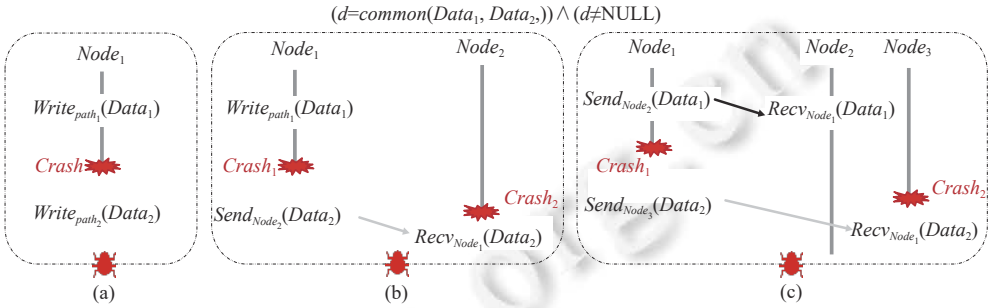


图 3 失效恢复缺陷中的共用数据写操作对 (→表示消息; 灰色项表示因节点失效而消失的项)

● 跨节点共用数据写操作相关失效恢复缺陷. 如图 3(b) 所示, 节点 $Node_1$ 的一个存储系统更新操作 ($Write_{path_1}$ ($Data_1$)) 和消息发送操作 ($Send_{Node_2}$ ($Data_2$)) 使用共用数据 d, 并将该共用数据存储在两个不同的位置 (存储路径 $path_1$ 和节点 $Node_2$). 因此 $Write_{path_1}$ 和 $Send_{Node_2}$ 是一对共用数据写操作. 在存储系统更新操作 $Write_{path_1}$ 完成之后, 使 $Node_1$ 在 $Send_{Node_2}$ 操作执行前失效 ($Crash_1$), 或使 $Node_2$ 在接收来自 $Node_1$ 的消息 ($Recv_{Node_1}$) 前失效 ($Crash_2$), 都会使节点 $Node_2$ 无法收到数据 d, 从而使存储在 $path_1$ 以及 (重启后的) $Node_2$ 节点上的数据状态不一致, 导致失效恢复缺陷. 需要注意的是, 这里的缺陷模式还考虑互换图示中存储系统更新操作和消息发送操作执行顺序的情况, 即先执行 $Send_{Node_2}$ ($Data_2$), 再执行 $Write_{path_1}$ ($Data_1$) 的情况.

● 分散节点共用数据写操作相关失效恢复缺陷. 如图 3(c) 所示, 一份共用数据 d 还可能被节点 $Node_1$ 的两个消息发送操作 $Send_{Node_2}$ ($Data_1$) 和 $Send_{Node_3}$ ($Data_2$) 分散到集群中其他两个节点 $Node_2$ 和 $Node_3$ 上. 类似的, 操作 $Send_{Node_2}$ 和 $Send_{Node_3}$ 也是一对共用数据写操作. 确保 $Send_{Node_2}$ 操作完成, 数据 d 被更新到节点 $Node_2$ 上之后, 使 $Node_1$ 在 $Send_{Node_3}$ 操作执行前失效 ($Crash_1$), 或使 $Node_3$ 在接收来自 $Node_1$ 的消息 ($Recv_{Node_1}$) 前失效 ($Crash_2$), 都会导致 (重启后的) $Node_3$ 节点和 $Node_2$ 节点状态不一致, 引发失效恢复缺陷.

根据我们总结的 3 个缺陷模式, 暴露失效恢复缺陷的关键是共用数据写操作对的识别, 以及在这些共用数据写操作对之间注入节点失效和节点重启来产生容易导致失效恢复缺陷的、不一致的系统状态.

1.3 技术挑战

我们需要解决如下两个主要挑战: (1) 哪些共用数据的不一致更易引发失效恢复缺陷? 追踪系统中每份数据的使用过程会引入巨大的性能开销, 并且只有关键数据的不一致才蕴含着失效恢复缺陷. (2) 如何识别使用共用数据的 I/O 写操作对? 一份关键数据可能在多个节点中多次传播, 要判断该数据是否最终被两个具有不同目标路径的 I/O 写操作所使用具有挑战.

为了解决以上两个挑战, Deminer 通过分析真实分布式系统和失效恢复缺陷来决定要追踪的关键数据类型, 例如从用户请求以及特定文件等中获得的用户数据和元数据, 从而在避免引入巨大性能开销的情况下尽可能考虑到分布式系统中所有容易引发错误的数据. 然后 Deminer 通过动态分布式系统数据追踪和离线执行轨迹分析来获得目标系统的一次执行中, 所有 I/O 写操作使用的关键数据来源. 基于分析结果, Deminer 可以把使用共用数据的两个 I/O 写操作识别为共用数据写操作对.

2 共用数据导向的失效恢复缺陷检测方法

Deminer 检测失效恢复缺陷的方法流程如图 4 所示. 首先, Deminer 追踪目标系统的一次正确执行. 在这个过程中, Deminer 决定要监测的关键数据, 动态追踪这些数据在目标系统中如何被使用, 并记录使用这些数据的 I/O 写操作及相关信息 (第 2.1 节). 其次, Deminer 通过记录的执行轨迹以及预定义的缺陷模式识别使用共用数据的 I/O 写操作对 (第 2.2 节). 再次, Deminer 预测可能造成不一致系统状态的节点失效注入点 (第 2.3 节). 最后, Deminer 测试所有的节点失效注入点, 并通过检查故障征兆来验证系统属性, 确认失效恢复缺陷 (第 2.4 节).

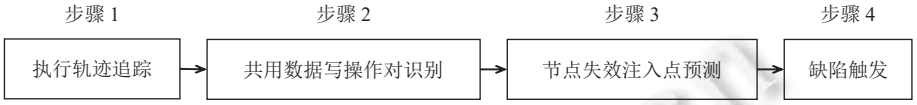


图 4 Deminer 方法流程

2.1 执行轨迹追踪

根据我们前面定义的缺陷模式, Deminer 需要监测目标系统中的关键数据 (主要是用户数据和元数据) 的使用, 动态追踪它们如何在分布式集群中传播, 以及这些数据是否最终被一个 I/O 写操作 (即存储系统写操作和消息发送操作) 所使用. 对每一个使用关键数据的 I/O 写操作, 我们记录相应的信息, 最终生成一组系统 I/O 写操作执行轨迹.

2.1.1 数据追踪

要判断两个 I/O 写操作是否使用共用数据, 我们需要分析每个 I/O 写操作所使用的数据来源. 我们通过动态污点追踪技术来追踪我们所关心的数据.

● 节点内数据追踪. Deminer 基于一个动态污点追踪系统 Phosphor[24]来对节点内数据传播进行追踪. 具体而言, 对于一个用污点标签标记的数据, 通过对运行在 JVM 内的所有字节码 (包括 JRE API) 进行插桩并追踪污点标签的传播, 来监测被标记数据的流向. 对运行在 JVM 中的每一个变量, 为其生成一个影子变量. 对每个对象 (object), 为该对象增加一个影子字段. 变量和对象相对应的污点标签存储在影子变量和影子字段中. 当把一个变量加载到 JVM 栈当中时, 该变量所对应的影子变量也被加载到栈中. 当进行计算时, 会把相应输入的污点标签进行结合. 对于图 2 所示的代码片段, 基于 Phosphor, Deminer 可以在 follower 节点识别到两个数据流: ②→⑥和④→⑤; 在 leader 节点识别到两个数据流: leader.epoch→①和 leader.epoch→③.

对于图 3 所示的缺陷模式, 仅通过节点内数据追踪, 只能识别到那些共用数据与 I/O 写操作位于同一节点的共用数据写操作. 如果两个 I/O 写操作使用的共用数据是由集群内其他节点传播而来, 则这样的共用数据写操作对无法被识别到. 例如, 对于图 2 所示的例子, 仅通过节点内数据追踪, 无法识别到 leader 和 follower 节点之间的数据流①→②和③→④.

● 跨节点数据追踪. 对跨节点传播的数据进行追踪, 关键是要将通过消息发送出去的数据在消息发送端和消息接收端进行匹配. 因此 Deminer 为每一个 RPC 调用以及 socket 消息生成一个随机数 (即消息 ID), 并通过为 RPC 调用以及 socket 消息增加额外的参数或字段来传播消息 ID. Deminer 会在消息发送端记录相应的消息 ID, 并在消息接收端用消息 ID 为收到的数据生成污点标签. 据此, Deminer 可以在离线分析阶段 (第 2.2 节) 进行跨节点数据流分析.

2.1.2 关键数据

对于一个分布式系统, 可能的数据来源, 也就是数据最初产生的地方, 主要包括用户请求、存储系统、物理设备的实时监测值、应用中的常量等. 而在这些数据源中, 只有关键数据的不一致才更容易引发失效恢复缺陷. 因此, 决定 Deminer 要追踪的关键数据至关重要.

一方面, 追踪系统中所有数据的使用情况是不现实的, 这会带来巨大的性能开销. 例如, 对本文实验中所用的目标系统和工作负载, 如果追踪所有数据的使用情况, 会造成内存溢出 (OOM), 系统无法运行. 进一步而言, 追踪

所有的数据也是不必要的. 对系统中所有数据进行追踪会使 Deminer 识别到大量不重要的共用数据写操作对. 例如, 一组具有相同消息类型的消息发送操作会由于使用了共用数据 (即消息类型值) 而被识别为一组共用数据写操作. 然而这些消息发送操作可能除了消息类型值以外没有其他共用数据, 在这些操作之间注入节点失效很可能不会引发失效恢复缺陷. 另一方面, 如果追踪的数据范围过小, 可能会导致遗漏重要的共用数据写操作对, 从而引入漏报. 而由开发者来人工指定每一个关键数据又是一项繁冗且容易出错的任务.

Deminer 的解决方法是在不引入巨大性能开销的情况下, 尽可能地对所有可能蕴含失效恢复缺陷的数据进行追踪. 我们通过观察真实分布式系统发现, 用户数据和元数据的不一致更易引发失效恢复缺陷. 其中用户数据指系统用户创建和拥有的数据, 例如用户创建的表格数据. 而元数据指用来表示系统状态的数据, 例如某个节点的状态信息、某个表格的存储位置等. 这些数据的不一致容易导致复杂的系统状态. 对这些不一致状态的不正确处理更容易引发严重的后果, 例如使用户读取到不一致的数据等. 具体而言, Deminer 考虑以下 4 类数据作为关键数据: (1) 从用户指定的存储路径, 包括本地文件、HDFS 文件和 ZooKeeper 节点 znode 中读到的数据. 这些存储目录下保存着用户数据和系统元数据; (2) 系统特定数据, 例如 nanoTime 和 currentTimeMillis, 这些数据常用作元数据的一部分, 如 Job ID; (3) 从用户请求中获得的数据, 这些数据往往包含用户关心的数据和配置信息, 例如 ZooKeeper 中的 create 请求, 会将用户所要创建的 znode 路径信息以及对应的 znode 值传给集群; (4) 从目标集群中其他节点发送的消息中获得的数据. Deminer 考虑此类数据是因为无法在节点间动态传播污点标签. 通过考虑这 4 类数据, Deminer 尽可能对分布式系统中的大多数数据来源进行追踪, 以减少缺陷漏报.

一旦关键数据被目标系统获取, Deminer 就会为其生成一个唯一的污点标签 (如表 1 所示) 来标记这份数据.

表 1　一个污点标签组成

污点标签字段	值
污点标签ID	long类型的数值
数据源	对于从存储系统中读到的数据, 用路径字符串来表示; 对于从消息从获得的数据, 用节点IP和消息ID的组合来表示
节点ID	生成该污点标签的节点IP和进程标识符 (PID)的组合

2.1.3　执行轨迹记录

基于我们的缺陷模式, Deminer 会为每一个使用关键数据的 I/O 写操作生成一条记录. 这些 I/O 写操作包括消息发送操作以及对存储系统 (即本地文件系统、分布式文件系统 HDFS 和分布式键值存储 ZooKeeper) 的创建/删除/更新操作.

一个直观地识别 I/O 写操作的方法是在 JRE 层插入代码来追踪用于发送消息和操作文件的 Java API 的使用. 然而, 在 JRE 层进行追踪, 难以获得应用于 HDFS 和 ZooKeeper 的 I/O 操作的相关信息, 例如存储路径. 在 JRE 层, 所有对 HDFS 和 ZooKeeper 的操作都会被识别为消息发送操作. 从一串消息字节中对路径字符串和存储数据进行区分十分困难. 此外, 对分布式系统而言, 为消息增加装饰信息 (如消息头) 或对消息进行加密是一种常见的方式. 在 JRE 层, 将原本的消息内容和装饰信息以及用于加密的数据进行区分十分困难. 这会导致大量消息发送操作会被识别为共用数据写操作. 例如, 许多消息用相同的数据进行加密.

因此, Deminer 选择在应用层追踪 RPC 调用和 socket 发送操作, 在应用层追踪对 HDFS 和 ZooKeeper 的操作, 以及在 JRE 层追踪本地文件相关操作. 对于 HDFS 和 ZooKeeper, Deminer 通过监测目标系统对它们的客户端 API 的使用来追踪相应的 I/O 操作, 并通过插桩代码来获取目标路径.

当一个 I/O 写操作使用关键数据时, Deminer 会为其生成一条如后文表 2 所示的执行轨迹记录.

2.2　共用数据写操作对识别

根据预定义的缺陷模式, 由同一个节点所执行的、具有不同目标路径 ID 且使用共用数据的 I/O 写操作对是一对共用数据写操作. Deminer 根据记录的系统执行轨迹, 比较每两个 I/O 写操作相对应的执行记录内容来判断这

两个 I/O 写操作是否为一对共用数据写操作. 其中, 判断两个执行记录是否使用共用数据, 一方面是通过直接比较两个记录所使用的污点标签交集是否为空来获得, 另一方面是基于污点标签以及消息 ID 来进行跨节点数据流分析, 比较两个记录是否共用由另一个节点获得的、经由消息传播到本地节点的数据所对应的污点标签. 具体的共用数据写操作对识别过程如算法 1 所示.

表 2 执行轨迹记录组成

执行轨迹记录字段	值
操作类型	存储系统I/O写操作或消息发送操作
目标路径ID	用路径字符串来表示存储系统写操作的目标路径 用目标节点IP和消息ID来表示RPC调用和socket发送操作的目标路径
污点标签	该操作所使用的关键数据对应的污点标签
调用栈	调用栈
时间戳	由RDTSCP 指令获得的纳秒级别本地时间戳计数
节点ID	执行该操作的节点IP和进程标识符 (PID)的组合
线程ID	执行该操作的线程哈希值

算法 1. 共用数据写操作对识别.

Input: *records* (Traced operation records)

1 **Function** isRelated(r_i, r_j):
2 $rst \leftarrow (r_i.des \neq r_j.des)$ and $(commonTaint(r_i, r_j) \neq \varnothing)$;
3 **return** *rst*;
4 **Function** commonTaint(r_i, r_j):
5 $com \leftarrow \varnothing$;
6 **if** $r_i.nodeId.equals(r_j.nodeId)$ **then**
7 $com \leftarrow intersection(r_i.taints, r_j.taints)$;
8 **if** $com = \varnothing$ **then**
9 $com \leftarrow remoteCommonTaint(r_i, r_j)$;
10 **end**
11 **end**
12 **return** *com*;
13 **Function** remoteCommonTaint(r_i, r_j):
14 $com \leftarrow \varnothing$;
15 **foreach** $t_i \in r_i.taints$ **do**
16 **foreach** $t_j \in r_j.taints$ **do**
17 $rmTs_i \leftarrow remoteTaints(t_i)$;
18 $rmTs_j \leftarrow remoteTaints(t_j)$;
19 $rmCom \leftarrow intersection(rmTs_i, rmTs_j)$;
20 $com.combine(rmCom)$;
21 **end**
22 **end**
23 **return** *com*;
24 **Function** remoteTaints(*t*):
25 $rst \leftarrow \varnothing$;

```
26  if t.source.isMsg() then
27      nodeIp ← t.source.nodeIp();
28      msgId ← t.source.msgId();
29      sendRec ← records.searchRecs(nodeIp, msgId);
30      if sendRec ≠ NULL then
31          rst ← sendRec.taints;
32      end
33  end
34  return rst;
```

算法 1 以第 2.1 节生成的 I/O 写操作记录 (即 records) 作为输入. 对于 records 中的任意两个 I/O 写记录 r_i 和 r_j, 如果他们有不同的目标路径并且使用共用数据, 则 Deminer 把他们识别为一对共用数据写操作 (第 1–3 行). 如第 2.1.3 节所述, Deminer 已经记录了每一个 I/O 写操作所使用的污点标签. 因此, 要识别两个操作是否使用共用数据, 只需要计算这两个操作所对应的记录是否有非空的污点标签交集 (第 4–12 行).

对于由同一个节点收集的记录 (第 6 行), 算法 1 首先直接计算这两个记录对应的污点标签的交集 (第 7 行). 如果它们有非空的污点标签交集, Deminer 则直接把这两个 I/O 写记录识别为一对共用数据写操作. 例如, 对于图 2 所示的例子, Deminer 会将操作①和操作③识别为一对使用共用数据 leader.epoch 的共用数据写操作.

如果两个记录的污点标签交集为空 (第 8–9 行), 例如图 2 中的⑤和⑥, Deminer 会进一步检查这两个记录是否使用了源自集群中另一个节点的共用数据 (第 13–23 行). 对每两个分别来自记录 r_i 和 r_j 的污点标签 (即来自 $r_i.taints$ 的 t_i 和来自 $r_j.taints$ 的 t_j), 算法 1 会尝试寻找它们相对应的远端污点标签, 也就是在另一个节点上生成并传播到当前节点的污点标签 (第 24–34 行). 随后, Deminer 计算这两个记录所对应的远端污点标签的交集 (第 17–19 行). 如果记录 r_i 和 r_j 具有非空的远端共用污点标签, 那么它们也是一对共用数据写操作.

给定一个污点标签, Deminer 基于该污点标签的数据源信息来寻找它对应的远端污点标签. 对于一个污点标签 t, 如果它的数据源是一个消息 (第 26 行), 这意味着 t 所对应的数据是由另一个节点通过一个消息传播而来. 算法 1 会根据发送该消息的节点 IP 和相应的消息 ID 来找到对应的消息发送操作 sendRec (第 27–29 行). 然后 Deminer 取得 sendRec 记录中的污点标签, 作为 t 的远端污点标签 (第 30–32 行). 例如, 图 2 中的操作⑤使用了在消息接收端④所生成的污点标签, 也就是使用了从 snapshot 消息 (③→④) 中收到的数据. 由于对应的消息发送操作③使用的污点标签包含 leader.epoch 数据的污点标签, 因此记录⑤所对应的远端污点标签是 leader.epoch 数据的污点标签. 类似地, Deminer 也可以识别到 leader 和 follower 节点间的另一条数据流: leader.epoch→①→②→⑥. 因此操作⑥的远端污点标签也是 leader.epoch 数据对应的污点标签. 由此可知, 操作⑤和操作⑥具有非空的远端污点标签, 它们也是一对共用数据写操作.

2.3 节点失效注入点预测

节点失效注入点表示应该在什么位置注入一个容易引发错误的节点失效事件. 根据预定义的缺陷模式, Deminer 会在两个共用数据写操作之间注入节点失效来使前一个 I/O 写操作被执行, 而后一个 I/O 写操作失败, 从而产生不一致的系统状态. Deminer 的目标是找到能测试更多不一致的系统状态以及更多失效恢复行为的节点失效注入点.

● 失效类型. 根据缺陷模式, Deminer 会生成两种类型的失效注入点: 本地失效和远端失效. 对于由节点 N 所执行的一对共用数据写操作, 一个本地失效指在执行这对共用数据写操作的节点 (即节点 N) 上注入节点失效. 例如图 3(a) 中的 $Crash$、图 3(b) 和图 3(c) 中的 $Crash_1$ 都是本地节点失效. 一个远端失效指使除节点 N 以外的一个节点失效, 如图 3(b) 和图 3(c) 中的 $Crash_2$. 本地节点失效和远端节点失效都可以打断共用数据写操作对的执行, 导致不一致的系统状态. 然而在不同节点上的失效会触发不同的失效恢复行为, 导致不同的影响. 因此, Deminer

需要同时考虑这两种失效类型.

● 失效注入点生成. 对于一个共用数据写操作对 (r_i, r_j), 其中 r_i 的时间戳小于 r_j 的时间戳, Deminer 按照以下 3 个步骤为其生成节点失效注入点. 首先, Deminer 会生成一个本地失效注入点 $[r_i, Crash_{\text{local}}, r_j]$, 也就是在前一个操作 r_i 执行之后, 后一个操作 r_j 即将执行之前注入一个本地失效事件. 接着, Deminer 会检查后一个操作 r_j 是否是一个向远端节点发送消息的消息发送操作. 如果是, Deminer 会为其生成一个远端节点失效注入点 $[r_i, Crash_{\text{remote}}, r_j]$, 使相应的消息接收节点在收到 r_j 发送的消息之前失效. 最后, 如果 r_i 和 r_j 位于两个不同的线程上, 且它们之间的时间窗口小于一个预定义的值, 那么这意味着 r_i 和 r_j 有可能是一对并发操作. 对这样的共用数据写操作对, Deminer 会试图翻转它们的执行顺序并生成相应的本地失效注入点和远端失效注入点: $[r_j, Crash_{\text{local}}, r_i]$ 和 $[r_j, Crash_{\text{remote}}, r_i]$.

对于一组具有相同失效类型、相同的后一个操作以及不需要翻转原来共用数据写操作对执行顺序的故障注入点, Deminer 只保留前一个操作具有最大时间戳的失效注入点. 这是因为为了避免引入错误和性能开销, Deminer 在缺陷触发阶段只控制与被测共用数据写操作对相关的 I/O 写操作的执行顺序, 并假设目标系统的其他 I/O 写操作在缺陷触发阶段和执行轨迹追踪阶段的执行序列相同. 对于一个节点失效注入点, Deminer 在前一个操作执行之后, 后一个操作将要执行之前注入节点失效. 因此具有相同的后一个操作的失效注入点蕴含着相同的不一致状态. 例如对两个具有相同失效类型 ($Crash_{\text{local}}$) 和相同后一个操作 (r_k) 的失效注入点 $[r_i, Crash_{\text{local}}, r_k]$ 和 $[r_j, Crash_{\text{local}}, r_k]$, 其中涉及的 3 个 I/O 写操作按照时间戳从小到大排序为 r_i 、 r_j 、 r_k . 在没有其他干扰因素的情况下, 在实际运行中这 3 个 I/O 写操作大概率会按照 r_i 、 r_j 、 r_k 的顺序被执行. 对于这两个失效注入点, Deminer 都会在 r_k 即将执行之前注入节点失效, 很可能会导致相同的不一致状态, 即 r_i 和 r_j 被执行, 而 r_k 未被执行, 从而引发相同的失效恢复行为. 因此, 对于这两个失效注入点我们只需要保留 $[r_j, Crash_{\text{local}}, r_k]$ 即可. 对于 r_i 被执行, 而 r_j 和 r_k 未被执行的情况, 如果 r_i 和 r_j 具有不同的目标路径, 则它们也是一对共用数据写操作对, Deminer 会通过在 r_i 和 r_j 之间注入节点失效来进行触发. 由于 Deminer 不控制所有 I/O 操作的执行顺序, Deminer 主要关注当前被测失效注入点中涉及的 I/O 操作的不一致所引发的失效恢复缺陷. 在未来的工作中, 我们可以通过控制更多的 I/O 操作来对系统进行更精确的控制, 从而可以检测到与多个共用数据写操作对相关的失效恢复缺陷, 减少缺陷漏报.

2.4 缺陷触发

Deminer 会自动测试每一个生成的失效注入点, 并通过观察故障征兆来确认一个失效恢复缺陷是否出现.

2.4.1 故障模型

由于节点失效和重启都会触发恢复行为, Deminer 对于每一个失效注入点会同时向目标系统注入一个节点失效事件以及相应的节点重启事件. 具体而言, 我们的故障模型如下所述. 在每次测试运行中, 只有一个节点失效注入点会被测试. 对一个失效注入点, 一个节点失效首先会被注入到目标节点中. 在随机等待一段时间后 (1 min 以内), 失效的节点会被重启. Deminer 之所以使用这种简单的故障模型, 是因为我们之前关于分布式系统中失效恢复缺陷的实证研究[9]表明不超过 1 个节点失效就可以触发分布式系统中 74% 的失效恢复相关缺陷. 另外, 我们的实证研究也表明大部分 (91.3%) 失效恢复缺陷的触发不需要特定的节点重启时机. 而分布式系统被期望至少要能够正确处理这种简单的故障模型.

2.4.2 故障注入

在缺陷触发阶段, Deminer 会启动一个故障注入控制器来控制整个测试过程. 在一次测试运行中, 在执行轨迹追踪阶段使用的工作负载会以触发模式执行. 而故障注入控制器会选择一个节点失效注入点进行测试, 并控制工作负载的启动和停止、故障的注入、故障征兆检查以及缺陷报告的生成. 以失效注入点 $[r_{\text{pre}}, Crash_{\text{local}}, r_{\text{latter}}]$ 为例, Deminer 会在每一个 I/O 写操作点收集系统运行时信息, 并检查一个节点是否运行到当前失效注入点. 当前一个操作 r_{pre} 已经被执行, 而后一个操作 r_{latter} 正要执行时, 该节点会向故障注入控制器报告并等待故障注入控制器的决策. 对于故障注入控制器收到的第一个报告, 它会直接使报告节点失效. 如果当前失效类型是远端失效, 即 $Crash_{\text{remote}}$, 故障注入控制器会从报告中获得远端节点 IP 使其失效, 并通知报告节点继续执行. 随后, 故障注入控

制器会重启之前失效的节点. 为了消除系统运行的不确定性影响, 对于一个被测的失效注入点, Deminer 最多会尝试测试 5 次, 直到相应的节点失效和重启被触发.

在故障注入过程中, 当失效注入点的后一个操作 r_{latter} 正要执行, 而前一个操作 r_{pre} 还没有被执行时, Deminer 会使后一个操作阻塞, 使其等待一段时间 (该等待时间可以根据相关超时数值配置) 直到前一个操作完成或等待时间耗尽. 然而, 对于需要翻转原来共用数据写操作对执行顺序的失效注入点, 例如 r_{pre} 的时间戳大于 r_{latter} 的时间戳, 那么后一个操作 r_{latter} 有可能会耗尽等待时间. 在这种情况下, Deminer 认为在 r_{latter} 和 r_{pre} 之间有可能存在一个 happens-before 关系, 也就是 r_{latter} 发生之后, r_{pre} 才会发生. Deminer 会记录这个可能的 happens-before 关系, 并将该失效注入点标记为不可执行. Deminer 会通过收集到的 happens-before 关系来跳过其他也可能无法执行的失效注入点. 由于等待时间设置不合理, 可能会造成不准确的 happens-before 关系分析, 从而导致缺陷漏报. 在未来的研究工作中, 我们可以通过更精确的 happens-before 关系分析方法来减少漏报.

2.4.3 故障征兆检查

对于每个被测的工作负载, 我们会根据领域知识, 人工书写特定的检查器来检测故障征兆. 具体而言, 我们的检查器会检查一般的故障征兆 (如运行日志中的 FATAL、ERROR 和 Exception, 以及节点崩溃), 也会检查操作特定的故障征兆 (例如操作返回错误代码或者读取到过时数据). 基于我们已经实现的检查器, 用户很容易就可以为其他工作负载实现特定检查器. 在实际使用中, 测试人员可以通过为特定目标系统下的特定工作负载设计更为精细的故障征兆检查器来减少缺陷漏报和缺陷误报.

对于操作特定的故障征兆 (如操作返回错误代码), Deminer 在工作负载运行过程中通过检查某个操作的运行返回结果来进行确认. 对于一般性故障征兆 (如节点宕机、错误日志等), Deminer 在节点失效和重启被注入、工作负载运行结束之后等待 15 s (该数值可配置) 再进行检查. 此时, 目标系统的所有恢复操作更可能都已完成, 从而可以检测系统的恢复结果是否出错. 此外, Deminer 还会在一次测试开始之后检测工作负载是否在特定运行时间之内 (平均运行时间的 2 倍) 完成来检测挂起错误.

3 方法实现

Deminer 被实现为一个命令行工具并且支持 JVM 1.8 版本. 使用 Deminer 主要需要 4 个步骤: (1) 通过运行一行命令生成一个 Java 运行环境的插桩版本; (2) 配置集群中的每一个节点使其使用插桩后的 JRE 并使用 Deminer 用于运行时插桩, 然后以执行轨迹追踪模式启动集群并运行一个工作负载; (3) 从每一个节点收集记录的执行轨迹, 然后通过运行一行命令来生成共用数据写操作对和节点失效注入点; (4) 配置目标系统以触发模式运行, 启动故障注入控制器来测试预测的失效注入点. 最后, 对于每一个没有通过检查器的失效注入点, Deminer 会为其生成一个缺陷报告.

Deminer 使用 ASM[25], 一个字节码操作库来对目标系统和 JRE 进行插桩. Deminer 支持 3 种存储系统: 本地文件系统、分布式文件系统 HDFS 以及分布式 key-value 存储 ZooKeeper. 对于本地文件读写操作, Deminer 追踪 FileInputStream、FileOutputStream 和 RandomAccessFile 这几个类中用于访存本地文件的本地 JDK 方法 (Java native method) 的使用. 对于 HDFS 和 ZooKeeper, Deminer 追踪目标系统对它们相应的客户端 API 的使用.

对于 RPC 调用, 最新的 Hadoop 和 HBase 版本基于 Protocol buffer 来实现 RPC. 对于 HBase RPC 接口, Deminer 通过检查接口类名是否以“Service$BolockingInterface”结尾来获得. 对于 Hadoop RPC 接口, Deminer 通过预先指定获得. 然后 Deminer 根据 RPC 接口可以很容易获得相应的 RPC 方法. 对于 socket, ZooKeeper 中的所有 socket 消息都继承了一个 Record 超类. Deminer 基于 Record 对象如何被使用来识别 socket 的发送和接收.

4 实验分析

我们在流行的开源分布式系统上对 Deminer 进行了验证, 来回答如下 3 个研究问题: (1) Deminer 能否有效检测分布式系统中的失效恢复缺陷? (2) Deminer 与其他节点失效注入方法比较如何? (3) Deminer 性能开销如何?

4.1 实验方法

4.1.1 目标系统

我们选择了 4 个流行的开源分布式系统对 Deminer 进行验证: 分布式文件系统 HDFS (HD); 分布式资源管理系统 YARN (YA); 分布式 NoSQL 数据库 HBase (HB); 分布式协调服务 ZooKeeper (ZK). 这 4 个分布式系统具有不同的功能, 并且实现了不同的失效恢复机制.

HDFS 使用复制机制将一个数据块的副本存储在多个 DataNode 上来容忍 DataNode 的失效. HDFS 也允许在一个集群中运行两个或多个冗余的 NameNode 节点来容忍 NameNode 节点的失效.

YARN 通过 ResourceManager 高可用性来避免单点失效问题. 此外, YARN 也提供配置允许 ResourceManager 在重启之后保持正常工作, 以及允许 NodeManager 在重启之后不丢失运行在该节点上的活动容器.

HBase 依靠 ZooKeeper 来维护配置信息、从节点 (RegionServer) 和客户端之间的交互以及分布式同步. 当一个 HMaster 节点失效后, 另一个备用的 HMaster 节点会被激活并接管集群. 当一个 RegionServer 节点失效后, 所有位于该节点上的数据分区 (即 region, HBase 上的最小存储和负载单元) 会被转移到另外一个 RegionServer 节点上. HBase 中的失效恢复机制使得 HBase 集群和集群中的数据在节点失效时仍然可以被访问.

ZooKeeper 在集群中大多数节点启动的情况下可以保持正常工作, 因此它可以容忍集群中少数节点宕机. ZooKeeper 中的每个节点都维护一个数据目录, 其中保存着快照文件 (snapshot) 和事务日志文件 (transactional log). 这些文件保存着一个 ZooKeeper 节点中的 znode 持久化副本. 节点重启后可以从这些文件中恢复数据.

4.1.2 失效恢复缺陷检测

为了验证 Deminer 在检测分布式系统失效恢复缺陷上的有效性, 我们把 Deminer 应用于目标系统的最新发布版本上. 对于 HBase, 它的版本 1 仍被支持和开发, 因此我们也在它的最新版本 1 上进行了测试. 表 3 列出了我们选择的目标系统版本. 如表 3 所示, 对于每个目标系统, 我们用常用的用户操作和管理操作设计了几个工作负载. 这些操作遵循自然顺序. 例如, 在向一个表更新数据之前先创建表.

表 3 目标系统实验设置

System	Workload
HDFS 3.3.1	1. Put/move file, read/write file 2. Read/write file + failover NameNode
YARN 3.3.1	1. Run WordCount
HBase 2.4.8/HBase 1.7.1	1. Create/read/update/truncate/delete table 2. Create/read/delete table + failover HMaster 3. Create/read/delete table + failover meta server
ZooKeeper 3.6.3	1. Create/read/update/delete znodes 2. Create/update znode + failover leader node

对于 HDFS, 我们设计了两个工作负载来测试文件系统操作和 NameNode 故障转移. HDFS 被配置使用 2 个副本, 使用 quorum journal manager (QJM) 来允许 HDFS 的高可用性, 并且基于 ZooKeeper 进行自动故障转移. 文件系统操作工作负载运行在一个拥有 2 个 NameNode 节点和 3 个 DataNode 节点的集群上. 而 NameNode 故障转移工作负载则运行在一个拥有 3 个 NameNode 节点和 3 个 DataNode 节点的集群上.

对于 YARN, 我们创建了一个工作负载来在拥有两个 ResourceManager 和两个 NodeManager 的节点上运行 WordCount 应用. YARN 集群被配置为 ResourceManager 高可用以及 ResourceManager 基于 ZooKeeper 进行保留工作模式的重启 (Work-preserving RM restart). 此外 YARN 集群运行在一个具有一个 NameNode 节点和一个 DataNode 节点的 HDFS 集群之上.

对于 HBase 1.7.1 (HB-1) 和 HBase 2.4.8 (HB-2), 我们设计了 3 个工作负载: 在一个拥有两个 HMaster 节点和两个 RegionServer 节点的集群上运行表管理操作; 在一个拥有 3 个 HMaster 节点和两个 RegionServer 节点的集群上运行 HMaster 故障转移; 在一个拥有两个 HMaster 节点和 3 个 RegionServer 节点的集群上运行 meta Region-

Server 故障转移. 每个 HBase 集群都运行在一个拥有 1 个 NameNode 节点和 1 个 DataNode 节点的 HDFS 集群和一个拥有 3 个节点的 ZooKeeper 集群上.

对于 ZooKeeper, 我们创建了两个工作负载: 在一个拥有 3 个节点的集群上运行 znode 共用数据写操作; 在一个拥有 5 个节点的集群上运行 leader 故障转移.

我们之前的实证研究工作[9]发现 97% 的失效恢复缺陷的触发, 只涉及不超过 4 个节点. 此外, 对于本文所提方法, 增加目标集群的规模会导致产生更多的 I/O 记录从而降低缺陷检查效率. 但对于同一工作负载, 扩大集群规模通常不会产生新的 I/O 类型, 因而不会影响缺陷检查结果. 因此, 在实验中, 我们采用了最小的集群规模来对 Deminer 进行验证. 以上所有工作负载在没有故障注入的情况下都能正确完成, 并且都被期望能从一个节点失效和一个节点重启中正确地恢复.

4.1.3 实验设置

我们用 Docker 19.03.3 在几台虚拟机上部署目标系统集群. 虚拟机中的操作系统使用 Ubuntu 20.04 和 JVM 1.8. 主机采用 64 位 CentOS Linux 7.3.1611 系统、JVM 1.8、两个 16 核 2.10 GHz Intel(R) Xeon(R) Gold 6130 CPU 和 125 GB RAM. 离线分析部分 (即共用数据写操作对识别和节点失效点预测) 以及故障注入控制器运行在主机上. 所有的轨迹追踪阶段和离线分析阶段的性能数值都是 3 次运行所得的平均值. 基准性能是在没有代码插桩的情况下度量得到的.

4.2 缺陷检测结果

4.2.1 总体实验结果

表 4 列出了 Deminer 所检测到的失效恢复缺陷的详细信息: JIRA 中的缺陷 ID (Bug ID 列)、该缺陷的故障征兆 (Failure symptom 列)、该缺陷的触发是否需要注入节点重启事件 (Reboot 列), 以及该缺陷能否被我们实验中的随机故障注入方法所触发 (Random 列).

表 4 Deminer 触发的失效恢复缺陷

Bug ID	Failure symptom	Reboot	Random
hb26370	Misleading error message	×	√
hb26391	Data staleness	×	×
hb26420	Cluster out of service	×	√
zk4283	Node downtime	√	×
zk4416	Node downtime	√	×
hd16381	Operation failure	×	×

Deminer 通过表 3 中所列的简单工作负载 (常用的用户操作和管理操作) 检测到 6 个失效恢复缺陷. 所有的缺陷都已经报告给开发者. 其中缺陷 zk4283 是一个之前已知的缺陷 (即第 1.1 节所示缺陷). 然而这个缺陷的修复并没有被合并到最新的 ZooKeeper 版本中. 另外在 HBase 1.7.1 中出现的缺陷 hb26420 之前没有被报告过, 但是在最新的 HBase 2.4.8 中已经被修复. 这些缺陷具有不同的故障征兆, 包括集群无服务、节点宕机、操作失败、数据过时以及误导性错误信息. 此外, 有两个缺陷 (zk4283 和 zk4416) 的触发需要注入节点重启事件.

4.2.2 误报分析

如表 5 所示, Deminer 总共报告了 52 个不同的测试故障 (Failures 列). 其中 7 个测试故障是由表 4 中列出的失效恢复缺陷所引发 (Bugs 列). 剩下的测试故障中包括 40 个误报 (False Pos. 列) 以及 5 个无法重现的故障 (Can't Repr. 列). 其中 7 个测试故障中, 两个分别来自 HB1 和 HB2 的测试故障具有相同的缺陷根因, 即 hb26370.

我们进一步研究了 Deminer 所产生的误报. 大多数误报 (38/40) 是由不合适的故障征兆检查器、预期的故障以及可以被系统容忍的故障导致. 例如, 在 HBase 的一个误报中, Deminer 向 meta RegionServer 注入了一个节点失效和重启. 随后在工作负载结束之后 Deminer 开始检查目标系统. 然而此时位于失效节点上的元数据表还没有被恢复, 使得该测试在检查器检查时, 由于没有可用的元数据表而使测试没有通过. 实际上, 在等待一段时间之后, 元数据表会被成功恢复. 在另一个 ZooKeeper 的误报中, follower 节点在与 leader 节点同步的过程中抛出了空指针异

常, 使得测试没有通过故障征兆检查器. 然而这个故障可以被集群所容忍. 该 follower 节点会重新进入 leader 选举阶段并重新与 leader 节点进行同步.

表 5 每个系统中出现的不同测试故障数

System	Failures	Bugs	False Pos.	Can't Repr.
HDFS	11	1	8	2
YARN	2	0	2	0
HB2	16	1*	15	0
HB1	15	3*	9	3
ZK	8	2	6	0
Total	**52**	**7**	**40**	**5**

注: *表示HB2中的一个故障和HB1中的一个故障具有相同的缺陷根因 (hb26370)

我们注意到虽然一些故障可以被集群容忍, 开发者也不确定对这些故障的处理是否会产生问题. 在另一个 HBase 的误报中, 测试由于遇到"TableNotFoundException: No state found for table"异常而没有通过检查器. HBase 集群通过忽略该异常来容忍这个故障. 然而, 开发者也不确定这样的处理是否正确, 并在代码中留下注释信息: "is it safe to just return false here?"

4.2.3 个例分析

我们通过缺陷 hb26420 来说明 Deminer 触发缺陷的过程. 在 hb26420 中, 当一个主节点 HM_1 成为活动主节点后, 它会选择一个 RegionServer 节点 RS_1 来存储元数据表, 并将元数据分区分配给 RS_1. 元数据表保存着集群中所有数据分区的位置信息, 因此当元数据表不可用时, HBase 集群无法正常工作. 当 RS_1 收到来自 HM_1 的打开元数据分区消息后, 它会执行相应的操作, 对 HDFS 和 ZooKeeper 进行一系列的更新, 最后将元数据分区的状态置为 OPENED. 当 RS_1 在元数据分区的分配过程中失效, HM_1 将会被一直阻塞, 等待元数据分区的分配完成. 然后整个集群都无法对外提供服务.

Deminer 成功识别到 RS_1 节点在元数据分区分配过程中的几对共用数据写操作. 这些共用数据写操作将同一份数据写入到 HDFS 和 ZooKeeper 的不同存储路径中. 在这些共用数据写操作对中间注入节点失效可以成功触发这个缺陷. 我们发现主节点 HM_1 在节点失效发生后, 已经通过 ZooKeeper 的通知检测到 RS_1 节点宕机. 然而, HM_1 却没有重启元数据分区分配过程. 这个缺陷在之前没有被报告过, 但在最新的 HBase 2.4.8 版本中得到了修复.

4.3 与其他故障注入方法比较

我们把 Deminer 和随机故障 (即节点失效) 注入方法进行了比较. 在随机故障注入方法中, 每个工作负载会被运行与 Deminer 预测的失效注入点数目相同的次数. 在每次测试运行中, 我们会随机选择目标集群中的一个节点注入节点失效及相应的节点重启. 我们在 0 和最长运行时长之间随机选择一个时间偏移值注入节点失效. 在随机等待 30 s 以内的时间后重新启动失效的节点. 如果一次测试运行在到达失效注入点前结束, 则节点失效和重启不会被注入到这次测试中. 我们在随机故障注入中使用和 Deminer 相同的检查器来检查故障征兆, 并通过分析运行日志来确认真实的缺陷.

表 6 列出了随机故障注入方法的实验结果. 其中 Test 列代表表 3 中相应的工作负载序号; Runs 列代表测试迭代次数; Triggered 列表示成功注入了节点失效和重启的测试次数; Time 列表示测试时间 (h); Known bugs 列表示随机故障注入方法检测到的缺陷数据, 其中括号中的数值为 Deminer 所检测到的缺陷数目. 我们发现 Deminer 比随机故障注入方法在检测失效恢复缺陷方面更加有效. 随机故障注入没有检测到任何新的缺陷, 触发了 Deminer 所检测到的 2 个缺陷. 这两个缺陷由在用户请求处理过程中和元数据分区分配过程中注入节点失效所触发. 这两个过程是相对耗时的, 因此随机故障注入可以在这个相对较大的缺陷触发时间窗口内注入节点失效.

我们没有把 Deminer 与别的故障注入方法进行比较. 这是因为其他故障注入方法没有公开可用的工具、难以使用或者被设计为检测其他类型的失效相关缺陷. 此外, 其他故障注入方法在没有复杂分析和建模的情况下也很难比随机故障注入表现更好.

表 6 随机故障注入测试结果

System	Test	Runs	Triggered	Time (h)	Known bugs
HDFS	1	874	775	41	0 (1)
	2	290	251	13	
YARN	1	412	355	40	0 (0)
HB2	1	1 191	679	94	0 (1)
	2	347	210	13	
	3	327	292	29	
HB1	1	1 171	879	56	2 (3)
	2	457	266	18	
	3	594	492	38	
ZK	1	292	181	4	0 (2)
	2	2 108	1 610	37	

4.4 性能分析

为了度量 Deminer 的性能开销, 我们在应用 Deminer 时记录了几个如表 7 所示的度量指标. 其中 Test 列代表表 3 中相应的工作负载序号; Baseline 列指在没有任何插桩的情况下原本的工作负载基准运行时间 (s).

表 7 Deminer 性能开销

System	Test	Baseline (s)	Tracing			Analysis		Triggering		
			Slowdown	Trace size (MB)	Records	Related pairs	Crashes	Runs	Triggered	Time (h)
HDFS	1	71	2.8×	7.4	254	1 499	874	1 353	166	92
	2	63	4.3×	4.5	212	401	290	728	146	54
YARN	1	36	8.6×	43.2	955	1 456	412	1 064	108	92
HB2	1	41	3.7×	3.5	237	3 989	1 191	490	297	41
	2	47	3.7×	2.9	201	1 042	347	264	153	26
	3	103	1.9×	1.9	180	972	327	220	131	37
HB1	1	44	2.5×	4.0	249	5 701	1 711	1 103	386	106
	2	37	3.3×	1.9	161	1 089	457	502	154	51
	3	121	1.7×	2.6	266	1 865	594	1 048	167	127
ZK	1	7	5.4×	0.8	96	154	292	226	196	8
	2	15	5.7×	1.6	362	2 835	2 108	3 783	983	264

在执行轨迹追踪阶段, 相比于基准运行时间, Deminer 在 YARN 上引入了 8.6 倍的性能开销. 而在其他系统上, Deminer 只引入了 1.7–5.7 倍的性能开销 (Slowdown 列). 引入性能开销的主要因素是污点标签的传播. 另外一个重要因素是我们在运行时对目标系统进行插桩. 如果我们提前通过静态方式对目标系统进行插桩可以使所费时长更短. Deminer 产生的执行轨迹记录在 0.8–43.2 MB 之间 (Trace size 列). 对于每个工作负载, 我们得到了 96–955 个使用了关键数据的 I/O 写记录 (Records 列).

不同工作负载下的离线分析阶段耗时都较小. 对于大多数测试, 离线分析部分都可以在 1 min 以内完成. 对于 YARN, 离线分析时间达到了 5 min. Deminer 识别到的共用数据写操作对在 154–5701 之间 (Related pairs 列), 而生成的节点失效注入点在 290–2108 之间 (Crashes 列).

在缺陷触发阶段, 测试迭代次数在 220–3783 之间 (Runs 列). 对于某些失效注入点的测试会耗尽最大尝试次数 (即 5 次). 而在触发阶段推测的 happens-before 关系会使得 Deminer 跳过对一些不可执行的失效注入点的测试. 成功触发的失效注入点的个数在 108–983 之间 (Triggered 列). 整个测试时长在 8–264 h 之间 (Time 列). 特别地, 在缺陷触发阶段, 每次测试运行的平均时长 (Time/Runs) 要大于执行轨迹追踪阶段的运行时长. 这是因为我们在缺陷触发阶段会注入节点失效和重启, 引发额外的恢复行为. 通过配置 Deminer 减少对每个失效注入点的最大测试尝试次数, 以及配置目标系统在失效检测过程中使用更小的超时时间, 可以减少测试时长.

考虑到分布式系统十分复杂, 以上的结果说明 Deminer 在应用于真实分布式系统方面具有有效性.

5 方法局限性讨论

在本节, 我们将讨论方法的局限性以及潜在的威胁.

(1) 共用数据写操作对识别. Deminer 的高效性和有效性取决于识别到的共用数据写操作对的质量. 第 1 个影响共用数据写操作对识别质量的因素是我们追踪的数据. 不是所有从消息和用户指定的存储路径中获得的数据的不一致都蕴含着失效恢复缺陷. 在未来的工作中, 我们考虑挖掘更有价值的数据来进行追踪. 第 2 个影响共用数据写操作对质量的因素是分布式系统数据流追踪的准确度. 基于消息 ID 的节点间数据流追踪是一种粗粒度的解决方案, 有可能引入误报. 此外, 缺乏本地文件系统的数据流追踪也会造成漏报. 我们将在未来的工作中提出更精确的数据流追踪方案.

(2) 节点失效注入点生成与触发. Deminer 目前只控制被测失效注入点中相关的 I/O 写操作的执行顺序, 这会导致缺陷漏报. 在未来的研究工作中, Deminer 可以通过控制所有 I/O 写操作的执行顺序以及一些其他的非确定性事件 (如线程创建、用户请求) 的执行顺序来更精确地控制目标系统的运行, 从而减少缺陷漏报. 另外, Deminer 通过在运行时等待一段时间的方式来推测可能的 happens-before 关系. 等待时间设置不合理也可能会造成缺陷漏报. 在未来的研究工作中, Deminer 可以通过追踪更多的事件来进行更准确的 happens-before 关系分析, 从而减少漏报.

(3) 缺陷检测. 对于一对共用数据写操作, 在它们之间注入节点失效可以导致不一致的系统状态. 然而这个不一致的状态可能被后续的恢复过程所容忍. 例如一些分布式系统 (如 HDFS) 以复制的方式来存储数据, 提高系统的可靠性. 在用于复制数据的几个共用数据写操作之间发生的节点失效通常可以被系统所容忍. 这是因为对于这类分布式系统, 容忍这样的节点失效是它们要考虑的一个主要问题. 在未来的工作中, 考虑通过影响预估分析来过滤掉一些可能被系统所容忍的故障注入点. 另外, 缺陷检测结果依赖于故障征兆检查器的质量. 在未来的工作中, 我们可以通过设计更为精细的故障征兆检查器来减少缺陷漏报和缺陷误报. 也可以通过更准确的目标系统运行时监测来识别系统恢复行为是否完成, 并在所有恢复行为完成后应用故障征兆检查器来减少缺陷漏报和缺陷误报.

对有效性的威胁方面, 我们在 4 个流行开源分布式系统的最新版本上对 Deminer 进行了验证. 然而, 实验结果可能无法反映 Deminer 在其他系统上的实验结果. 但是, 通过选择具有不同功能 (即分布式文件系统、分布式资源协调系统、分布式 NoSQL 数据库和分布式同步服务) 和各种恢复恢复机制 (例如自动故障转移、同步、备份等) 的分布式系统, 力求在系统选择上做到不偏不倚.

6 相关研究讨论

(1) 故障注入. 故障注入是分布式系统中暴露缺陷的一种常用技术. 故障注入框架例如 Chaos Monkey[11]和 Jepsen[12]通过随机注入节点失效和其他类型的故障来通过多次运行触发缺陷. PreFail[26]允许测试者自定义故障注入策略来减少故障注入空间. NEAT[27]为开发者提供简单的 API 来使开发者能够注入和修复网络分区故障. 然而这些方法难以暴露对节点失效和节点重启具有特殊时间要求的失效恢复缺陷.

最近的工作也提出基于协议感知或领域知识来进行故障注入. CORDS[28]通过每次在一个节点上、一个文件系统数据块中注入一个故障来测试一个分布式系统是否可以从文件系统故障中正确地恢复. CoFI[29]在不一致的系统状态上注入网络分区故障, 并控制注入网络分区故障的起始点和结束点. PACE[30]关注相关的崩溃漏洞, 即当特定数据碎片的所有副本同时崩溃时, 用户级别的保证是否会被违反. CrashTuner[19]在节点访问元信息变量时注入节点失效. 它通过一个基于日志的静态程序分析技术来推断元信息变量. 缺陷检测结果依赖于日志质量. 这些工作与 Deminer 关注不同的故障场景.

(2) 模型检查. 分布式系统模型检查器[13−17,31,32]也可以用于检测与节点失效相关的缺陷. 这些工作拦截包括节点失效在内的不确定性分布式事件, 并对它们的执行顺序进行重新排序. 然而, 这些分布式系统模型检查器不止关注与节点失效相关的缺陷. 因此, 它们在暴露一个失效恢复缺陷之前, 会探索大量与节点失效无关的状态空间. 所有的分布式系统模型检查器在应用于真实的分布式系统时都会有状态空间爆炸问题.

(3) 分布式系统缺陷检测. FCatch[18]通过观察节点失效发生时可能的冲突操作来预测与故障时机相关的缺陷. ALICE[33]通过对特定文件系统上可能发生的所有崩溃状态进行建模, 并检查应用程序是否能够从这些崩溃状态中正确恢复, 来发现应用程序中的崩溃漏洞. 另外还有一些工作检测分布式系统中与节点失效无关的缺陷, 包括分布式并发缺陷[34]、性能级联缺陷[35]、与数据损坏相关的挂起错误[36]、实现错误的异常处理程序或缺少异常处理程序[37,38]等. 这些工作解决和 Deminer 正交的问题.

(4) 验证. 许多分布式协议 (例如 Paxos[39]、Raft[40]和 PBFT[41]) 提供了书面的证明. 也有一些技术提出了机器可检查的证明[42]. 部分工作研究通过 Coq[43]和 TLA[44]等证明框架来建立可验证的分布式系统协议[45−47]. 然而, 这些经过形式化验证的分布式系统实现依然可能存在缺陷[48]. 最新的技术离构建在性能和规模上与现有系统相匹配的、可验证的分布式系统还很遥远.

7 总 结

失效恢复缺陷影响分布式系统的可用性和可靠性, 而此类缺陷往往只有节点失效发生在特定时机时才会被触发, 因此检测此类缺陷具有挑战性. 本文提出了一种共用数据导向的分布式系统失效恢复缺陷检测方法 Deminer. Deminer 通过动态追踪关键数据在运行时的使用来识别使用共用数据的相关 I/O 写操作对, 并在这些共用数据写操作对之间自动注入节点失效和节点重启来触发失效恢复缺陷. Deminer 已经在 4 个流行的开源分布式系统上检测到 6 个失效恢复缺陷. 我们的实验表明 Deminer 在失效恢复缺陷检测方面具有有效性.

References:

[1] Chang F, Dean J, Ghemawat S, Hsieh WC, Wallach DA, Burrows M, Chandra T, Fikes A, Gruber RE. Bigtable: A distributed storage system for structured data. ACM Trans. on Computer Systems, 2008, 26(2): 4. [doi: 10.1145/1365815.1365816]

[2] DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: Amazon's highly available key-value store. In: Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles. Stevenson: ACM, 2007. 205–220. [doi: 10.1145/1294261.1294281]

[3] Ghemawat S, Gobioff H, Leung ST. The Google file system. In: Proc. of the 19th ACM Symp. on Operating Systems Principles. Bolton Landing: ACM, 2003. 29–43. [doi: 10.1145/945445.945450]

[4] Hindman B, Konwinski A, Zaharia M, Ghodsi A, Joseph AD, Katz R, Shenker S, Stoica I. Mesos: A platform for fine-grained resource sharing in the data center. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2011. 295–308.

[5] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B. Apache hadoop YARN: Yet another resource negotiator. In: Proc. of the 4th Annual Symp. on Cloud Computing. Santa: ACM, 2013. 5. [doi: 10.1145/2523616.2523633]

[6] Dean J, Ghemawat S. MapReduce: Simplified data processing on large clusters. In: Proc. of the 6th Symp. on Operating Systems Design and Implementation. San Francisco: USENIX Association, 2004. 137–150.

[7] Burrows M. The chubby lock service for loosely-coupled distributed systems. In: Proc. of the 7th Symp. on Operating Systems Design and Implementation. Seattle: USENIX Association, 2006. 335–350.

[8] Dean J. Designs, lessons and advice from building large distributed systems. 2009. http://iepg.org/iepg/2009-11-ietf76/dean-keynote-ladis2009.pdf

[9] Gao Y, Dou WS, Qin F, Gao CS, Wang D, Wei J, Huang RR, Zhou L, Wu YM. An empirical study on crash recovery bugs in large-scale distributed systems. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena: ACM, 2018. 539–550. [doi: 10.1145/3236024.3236030]

[10] Zhang J, Zhang C, Xuan JF, Xiong YF, Wang QX, Liang B, Li L, Dou WS, Chen ZB, Chen LQ, Cai Y. Recent progress in program analysis. Ruan Jian Xue Bao/Journal of Software, 2019, 30(1): 80−109 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

[11] Chaos monkey. 2017. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey

[12] Jepsen. 2018. https://github.com/jepsen-io/jepsen

[13] Leesatapornwongsa T, Hao MZ, Joshi P, Lukman JF, Gunawi HS. SAMC: Semantic-aware model checking for fast discovery of deep

bugs in cloud systems. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 399–414.

[14] Lukman JF, Ke H, Stuardo CA, Suminto RO, Kurniawan DH, Simon D, Priambada S, Tian C, Ye F, Leesatapornwongsa T, Gupta A, Lu S, Gunawi HS. FlyMC: Highly scalable testing of complex interleavings in distributed systems. In: Proc. of the 14th European Conf. on Computer Systems. Dresden: ACM, 2019. 1–16.

[15] Anand V. Dara: Hybrid model checking of distributed systems. In: Proc. of the 26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Lake Buena Vista: ACM, 2018. 977–979. [doi: 10.1145/3236024.3275438]

[16] Yang JF, Chen TS, Wu M, Xu ZL, Liu XZ, Lin HX, Yang M, Long F, Zhang LT, Zhou LD. MODIST: Transparent model checking of unmodified distributed systems. In: Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2009. 213–228.

[17] Guo HY, Wu M, Zhou LD, Hu G, Yang JF, Zhang LT. Practical software model checking via dynamic interface reduction. In: Proc. of the 23rd ACM Symp. on Operating Systems Principles. Cascais: ACM, 2011. 265–278. [doi: 10.1145/2043556.2043582]

[18] Liu HP, Wang X, Li GP, Lu S, Ye F, Tian C. FCatch: Automatically detecting time-of-fault bugs in cloud systems. In: Proc. of the 23rd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Williamsburg: ACM, 2018. 419–431. [doi: 10.1145/3173162.3177161]

[19] Lu J, Liu C, Li L, Feng XB, Tan F, Yang J, You L. CrashTuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis. In: Proc. of the 27th ACM Symp. on Operating Systems Principles. Huntsville: ACM, 2019. 114–130. [doi: 10.1145/3341301.3359645]

[20] Apache ZooKeeper™. 2020. http://zookeeper.apache.org

[21] Apache HBase. 2019. https://hbase.apache.org

[22] Apache hadoop YARN. 2022. https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html

[23] HDFS architecture guide. 2022. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[24] Bell J, Kaiser G. Phosphor: Illuminating dynamic data flow in commodity JVMs. In: Proc. of the 2014 ACM Int'l Conf. on Object Oriented Programming Systems Languages & Applications. Portland: ACM, 2014. 83–101. [doi: 10.1145/2660193.2660212]

[25] ASM. 2022. https://asm.ow2.io/

[26] Alvaro P, Rosen J, Hellerstein JM. Lineage-driven fault injection. In: Proc. of the 2015 ACM SIGMOD Int'l Conf. on Management of Data. Melbourne: ACM, 2015. 331–346. [doi: 10.1145/2723372.2723711]

[27] Alquraan A, Takruri H, Alfatafta M, Al-Kiswany S. An analysis of network-partitioning failures in cloud systems. In: Proc. of the 13th USENIX Conf. on Operating Systems Design and Implementation. Carlsbad: USENIX Association, 2018. 51–68.

[28] Ganesan A, Alagappan R, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to single errors and corruptions. In: Proc. of the 15th USENIX Conf. on File and Storage Technologies. Santa Clara: USENIX Association, 2017. 149–165.

[29] Chen HC, Dou WS, Wang D, Qin F. CoFI: Consistency-guided fault injection for cloud systems. In: Proc. of the 35th IEEE/ACM Int'l Conf. on Automated Software Engineering. Melbourne: IEEE, 2020. 536–547.

[30] Alagappan R, Ganesan A, Patel Y, Pillai TS, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Correlated crash vulnerabilities. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 151–167.

[31] Simsa J, Bryant R, Gibson G. dBug: Systematic evaluation of distributed systems. In: Proc. of the 5th Int'l Conf. on Systems Software Verification. Voncouver: USENIX Association, 2010. 3.

[32] Killian C, Anderson JW, Jhala R, Vahdat A. Life, death, and the critical transition: Finding liveness bugs in systems code. In: Proc. of the 4th USENIX Conf. on Networked Systems Design & Implementation. Cambridge: USENIX Association, 2007. 18.

[33] Pillai TS, Chidambaram V, Alagappan R, Al-Kiswany S, Arpaci-Dusseau AC, Arpaci-Dusseau RH. All file systems are not created equal: On the complexity of crafting crash-consistent applications. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 433–448.

[34] Liu HP, Li GP, Lukman JF, Li JX, Lu S, Gunawi HS, Tian C. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In: Proc. of the 22nd Int'l Conf. on Architectural Support for Programming Languages and Operating Systems. Xi'an: ACM, 2017. 677–691. [doi: 10.1145/3037697.3037735]

[35] Li JX, Chen YX, Liu HP, Lu S, Zhang YM, Gunawi HS, Gu XH, Lu XC, Li DS. Pcatch: Automatically detecting performance cascading bugs in cloud systems. In: Proc. of the 13th EuroSys Conf. Porto: ACM, 2018. 7. [doi: 10.1145/3190508.3190552]

[36] Dai T, He JZ, Gu XH, Lu S, Wang PP. DScope: Detecting real-world data corruption hang bugs in cloud server systems. In: Proc. of the 2018 ACM Symp. on Cloud Computing. Carlsbad: ACM, 2018. 313–325. [doi: 10.1145/3267809.3267844]

[37] Yuan D, Luo Y, Zhuang X, Rodrigues GR, Zhao X, Zhang YL, Jain PU, Stumm M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In: Proc. of the 11th USENIX Conf. on Operating Systems Design and Implementation. Broomfield: USENIX Association, 2014. 249–265.

[38] Saha S, Lozi JP, Thomas G, Lawall JL, Muller G. Hector: Detecting resource-release omission faults in error-handling code for systems software. In: Proc. of the 43rd Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks. Budapest: IEEE, 2013. 1–12. [doi: 10.1109/DSN.2013.6575307]

[39] Lamport L. The part-time parliament. ACM Trans. on Computer Systems, 1998, 16(2): 133–169. [doi: 10.1145/279227.279229]

[40] Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proc. of the 2014 USENIX Conf. on USENIX Annual Technical Conf. Philadelphia: USENIX Association, 2014: 305–320.

[41] Castro M, Liskov B. Practical Byzantine fault tolerance. In: Proc. of the 3rd Symp. on Operating Systems Design and Implementation. New Orleans: USENIX Association, 1999. 173–186.

[42] Lu TX. Formal verification of the pastry protocol using TLA$^+$. In: Proc. of the 1st Int'l Symp. on Dependable Software Engineering: Theories, Tools, and Applications. Nanjing: Springer, 2015. 284–299. [doi: 10.1007/978-3-319-25942-0_19]

[43] The Coq proof assistant. 2021. https://coq.inria.fr/

[44] The TLA+ home page. 2022. https://lamport.azurewebsites.net/tla/tla.html

[45] Deligiannis P, Donaldson AF, Ketema J, Lal A, Thomson P. Asynchronous programming, analysis and testing with state machines. In: Proc. of the 36th ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 154–164. [doi: 10.1145/2737924.2737996]

[46] Hawblitzel C, Howell J, Kapritsos M, Lorch JR, Parno B, Roberts ML, Setty S, Zill B. IronFleet: Proving practical distributed systems correct. In: Proc. of the 25th Symp. on Operating Systems Principles. Monterey: ACM, 2015. 1–17. [doi: 10.1145/2815400.2815428]

[47] Lesani M, Bell CJ, Chlipala A. Chapar: Certified causally consistent distributed key-value stores. In: Proc. of the 43rd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. St. Petersburg: ACM, 2016. 357–370. [doi: 10.1145/2837614.2837622]

[48] Fonseca P, Zhang KY, Wang X, Krishnamurthy A. An empirical study on the correctness of formally verified distributed systems. In: Proc. of the 12th European Conf. on Computer Systems. Belgrade: ACM, 2017. 328–343. [doi: 10.1145/3064176.3064183]

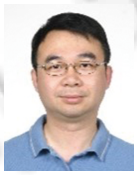
附中文参考文献:

[10] 张健, 张超, 玄跻峰, 熊英飞, 王千祥, 梁彬, 李炼, 窦文生, 陈振邦, 陈立前, 蔡彦. 程序分析研究进展. 软件学报, 2019, 30(1): 80–109. http://www.jos.org.cn/1000-9825/5651.htm [doi: 10.13328/j.cnki.jos.005651]

高钰(1992－), 女, 博士, 主要研究领域为软件工程, 系统测试, 系统可靠性.

窦文生(1984－), 男, 博士, 副研究员, CCF 专业会员, 主要研究领域为程序分析, 软件工程.

王栋(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析, 软件可靠性.



魏峻(1970－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为软件工程, 网络分布式计算.

戴千旺(1998－), 男, 硕士, 主要研究领域为软件可靠性, 数据库系统测试.