软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

分布式系统动态测试技术研究综述*

陈元亮[1], 马福辰[1], 周远航[1], 颜臻[1], 姜宇[1], 孙家广[1]

[1](清华大学 软件学院,北京 100084)

通讯作者: 姜宇, E-mail: jiangyu198964@126.com

摘 要: 分布式系统是当今计算生态系统的支柱，它使得现代计算更加强大、可靠和灵活，覆盖了从云计算、大数据处理到物联网等多个关键领域。然而，由于系统的复杂性，分布式系统在代码实现过程中总是会不可避免地引入一些代码缺陷，从而对系统的可用性、鲁棒性以及安全性造成巨大威胁。因此，分布式系统的测试以及缺陷挖掘工作十分重要。动态测试技术在系统运行中进行实时分析，以挖掘其缺陷，评估其行为和功能，被广泛用于各种系统应用的缺陷检测中，并成功发现了许多代码缺陷。本文首先提出了分布式系统四层缺陷威胁模型，并基于它分析了分布式系统测试需求与主要挑战，提出了对分布式系统进行动态测试的一般框架；接着从挖掘不同类型系统缺陷的角度介绍了典型的分布式系统动态测试工具；然后总结了包括不同维度测试输入生成、系统关键状态感知、缺陷判定准则构建在内的分布式动态测试的关键技术；并对当前主流分布式系统动态测试工具的覆盖率和缺陷发现能力进行了评估，从初步实验结果中我们可以看出多维度测试输入技术能有效提高分布式系统测试效率。最后，本文讨论了分布式系统动态测试的新趋势以及可能的未来发展方向。

关键词: 分布式系统; 动态测试; 缺陷挖掘; 模糊测试; 故障注入; 综述

中图法分类号: TP311

中文引用格式: 陈元亮,马福辰,周远航,颜臻,姜宇,孙家广.分布式系统动态测试技术研究综述.软件学报, 2024,32(7). http://www.jos.org.cn/1000-9825/0000.htm

英文引用格式: Chen YL, Ma FC, Zhou YH, Yan Z, Jiang Y, Sun JG. A Survey of Dynamic Testing Methods for Distributed Systems. Ruan Jian Xue Bao/Journal of Software, 2024 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

A Survey of Dynamic Testing Methods for Distributed Systems

Chen Yuan-Liang[1], Ma Fu-Chen[1], Zhou Yuan-Hang[1], Yan Zhen, Jiang Yu[1], Sun Jia-Guang[1]

[1](School of Software, Tsinghua University, Beijing 100084, China)

Abstract: Distributed systems underpin modern computing, enabling powerful, reliable, and flexible operations across domains such as cloud computing, big data, and IoT. However, their complexity often leads to code defects that threaten usability, robustness, and security, making testing and defect detection essential. Dynamic testing, which evaluates systems during runtime, plays a key role in uncovering defects and assessing functionality. This paper introduces a four-layer bug threat model for distributed systems, covering system configuration, user requests, node communication, and environmental faults. Based on this model, it analyzes the challenges of testing distributed systems and proposes a general framework for dynamic testing. The paper highlights critical techniques such as multidimensional test input generation, system-critical state awareness, and defect judgment criteria. Additionally, the paper reviews popular dynamic testing tools and evaluates their effectiveness in defect discovery and test coverage. The findings show that multidimensional input generation significantly enhances testing efficiency. Finally, the paper discusses emerging trends and future directions in dynamic testing of distributed systems, aiming to address their inherent challenges and improve testing outcomes.

Key words: Distributed System; Dynamic Testing; Bug Detection; Fuzzing; Fault Injection; Survey

* 基金项目：国家重点研发计划(2022YFB3104000);
Foundation item: National Key Research and Development Project (2022YFB3104000);
收稿时间: 2024-08-20; 修改时间: 2024-11-25; 采用时间: 2024-12-02

随着数据和计算向云端迁移，大规模分布式系统已成为现代计算的核心，被广泛应用于分布式存储、计算框架和云原生微服务架构等领域[1][2]。典型代表包括云计算平台（如 AWS、Google Cloud）、大数据处理系统（如 Hadoop、Spark）、分布式存储（如 GFS、HDFS）、云原生系统（如 Kubernetes、Istio）及区块链系统（如 Ethereum、Bitcoin）。相比传统单机系统，分布式系统具备显著的可扩展性和容错能力，以更低成本提供强大计算能力，已经成为支撑大规模网络应用的基石[3]。

然而，开发者在分布式系统实现过程中难免会引入代码缺陷（bug），这些缺陷不仅影响系统的正常运行，还对性能、可靠性和安全性构成威胁。例如，Facebook 平台因一个缺乏正确超时检查的缺陷导致大规模服务节点宕机，影响了自身和第三方应用，造成巨大经济损失[4]。类似地，ZOOKEEPER-3189 缺陷导致 ZooKeeper 节点长时间无法响应客户端请求，导致服务中断数小时[5]。此类缺陷还可能被攻击者利用，发动分布式拒绝服务（DDoS）攻击，进一步加剧经济损失。因此，分布式系统的高效测试至关重要，不仅有助于保障系统的可用性和稳定性，还能增强整个生态系统的安全性。

目前已有大量的工作来提升软件系统的缺陷检测与测试效率，包括模型检测[6]、静态分析[7]、动态测试[8]等技术。模型检测技术通过探索属性状态空间，判断系统模型是否满足给定的规范，从而进行测试分析并挖掘其中的缺陷。常用模型检测工具如 NuSMV[9]、PRISM[10]、SPIN[11]等，在传统软件测试中已经发现了许多缺陷。然而，面对分布式系统的大规模代码，模型检测常遇到状态空间爆炸等问题，难以取得理想效果。静态分析技术通过对代码结构的分析来挖掘潜在缺陷，而无需执行程序。典型静态分析工具如 CORBA[12]、SonarQube[13]、Coverity[14]等在分布式系统中已被广泛应用，但静态分析面临误报率高的挑战。动态测试技术则是在软件系统运行过程中生成大量测试输入，并通过实时监控系统的运行状态来检测异常行为或系统缺陷。与其他技术相比，它能够有效缓解模型检测中的状态空间爆炸问题，且误报率较低，因此被广泛应用于分布式系统的缺陷挖掘过程中[15][16][17]。然而相比于传统软件的动态测试，分布式系统动态测试具有测试环境复杂、输入维度繁多、行为状态多变、缺陷种类多样等特点。因此，近年来针对分布式系统不同维度的输入、不同类型的系统缺陷，诞生了许多动态测试技术研究成果，累计发现了大量的分布式系统缺陷，有效提升了分布式系统的可用性及安全性[18][19][20][21]。

分布式系统的动态测试是其安全研究的重要方向，近年来相关研究与综述不断涌现。Lima 等人[22]在 2016 年对 147 名软件测试专业人员进行了调研，强调了分布式系统测试的重要性及对自动化工具的迫切需求。Lahami 等人[23]则探讨了系统动态变化和更新过程中运行时测试的重要性。然而，现有综述缺乏对分布式系统动态测试的关键技术以及主要挑战进行全面分析与总结。本文首先在第一章提出了分布式系统的四层缺陷威胁模型，并基于此模型分析总结了动态测试的主要需求和挑战，提出了通用的动态测试框架。第二章介绍了针对分布式系统功能性、安全性、一致性、性能、鲁棒性及扩展性缺陷的典型测试工具。第三章分析了近年来在四维度测试输入生成、系统关键状态感知、缺陷判定准则构建等方面的研究进展。第四章通过覆盖率和缺陷发现能力评估了当前主流动态测试工具，实验结果显示这些工具的覆盖率仍然较低，许多深层路径缺陷未被检测到，而多维度测试输入生成技术可以有效提升测试效率。第五章讨论并展望了分布式系统测试未来的研究方向。

1 分布式系统动态测试概述

1.1 动态测试技术

动态测试技术通过在特定测试输入下运行程序并分析其行为或输出来发现缺陷[24][25][26][27]，其核心流程如图 1 所示，主要包含三个重要的步骤：测试输入生成，程序执行以及缺陷检测。其中，测试输入生成器依据生成策略，生成待测程序输入。程序执行是指待测程序执行前一步骤生成的测试输入。缺陷监视器主要包含缺陷判定器和判定准则，用以判定程序是否触发缺陷。

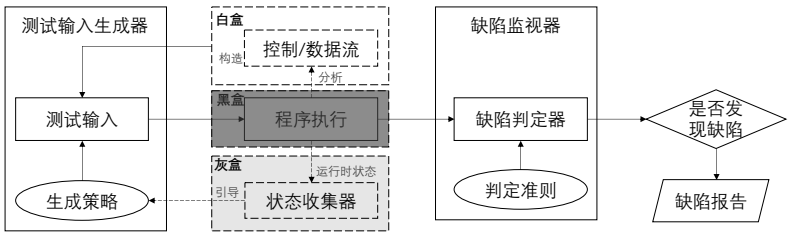


图 1 动态测试一般流程

测试输入生成：动态测试的核心在于生成一系列能够触发程序潜在故障的测试输入。这些输入不仅需要满足程序的基本格式要求，还要具有足够的“问题性”以提高揭露缺陷的可能性。高效生成能引发程序错误的测试输入是动态测试工具研究的关键。测试输入生成策略主要分为两类：基于生成（generation-based）和基于变异（mutation-based）。基于生成的方法利用目标程序的语法建模，生成大量符合格式要求的输入；基于变异的方法通常采用遗传算法，通过对初始输入进行变异，产生新的输入以尽可能探索程序的状态。

程序执行：在测试输入生成之后，这些用例会被输入到目标程序中进行测试。动态测试工具自动启动目标程序，执行测试过程，并监控和控制测试输入的处理，保障测试流程顺利进行。根据对目标程序的控制粒度，测试方法可分为黑盒测试、白盒测试和灰盒测试三种。黑盒测试将测试对象视为一个黑盒子，不考虑程序内部逻辑，仅依据需求说明进行功能验证。白盒测试则分析代码的内部结构（如控制流和数据流）及其背后的逻辑，要求测试人员具备编码能力并了解测试的软件。灰盒测试介于两者之间，既考虑程序的内部结构也考虑外部功能，使用部分运行时信息动态引导测试输入生成，以提高测试效率和覆盖率。

缺陷检测：在执行目标程序时，缺陷监视器实时监控程序运行状态，判断是否违背预定义的判定准则。一旦发现违规，即判定为程序缺陷，记录并捕获违规行为。捕获后，动态测试工具会将相应的测试输入存储以供重放和分析使用。监视器检测的对象有两种类型：（1）行结果审查：将程序运行输出与预知输出对比，分析是否存在缺陷。这种方法直观，但无法检测到未反映在输出结果中的缺陷；（2）执行过程检查：通过插桩或其他监控技术分析程序运行行为，查找错误行为。此方法能更直接观察到缺陷的触发，即使未导致错误输出，但需要提前定义错误行为。

1.2 分布式系统的缺陷威胁模型

分布式系统是由多个独立的计算节点通过网络协作完成共同任务的计算系统。一个典型分布式系统的运行流程如下图 2 所示，主要包括四个关键步骤：（1）根据系统配置文件启动分布式网络；（2）客户端发起请求，由分布式服务进行响应；（3）在分布式节点间协同完成请求的处理；（4）整个过程中，系统在分布式环境下软硬件实时进行交互，保障分布式系统高效与稳定性的兼顾。

与传统程序相比，分布式系统具有更复杂的系统配置、更丰富的服务功能、更频繁的节点通信以及更复杂的环境交互等特点。这些特征不仅增加了系统开发、维护和管理的难度，还大大提高了代码实现过程中引入缺陷的可能性，从而导致分布式系统面临更多的缺陷威胁。因此，在对其进行测试时，我们需要关注更多的输入维度和缺陷类型。

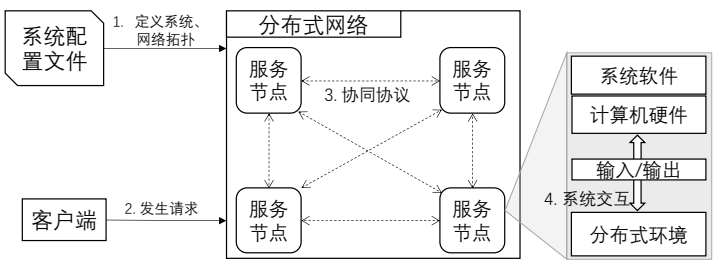


图 2 分布式系统一般运行流程

本文基于图 2 的分布式系统一般运行流程，根据不同关键流程中可能出现的系统缺陷进行分类，建立了四层缺陷威胁输入模型：系统配置处理缺陷、用户请求处理缺陷、节点间消息处理错误以及环境交互故障。分布式系统四层威胁模型及其常见缺陷如表 1 所示。

表 1 分布式系统威胁模型与常见缺陷

威胁输入层面	威胁详细描述	常见缺陷
系统配置处理层面	为了提高分布式系统的灵活性和可扩展性，设计了各种系统配置。随着分布式系统规模和复杂性的增加，系统配置变得愈加复杂，相应的代码处理逻辑也更容易出错。研究表明，分布式系统中大多数的错误和缺陷与业务逻辑代码无关，而是由不恰当的系统配置处理逻辑引起的[28][29][30]。	配置语义不正确 配置更新失败 配置兼容错误 配置权限错误 配置依赖错误 冗余配置项 … …
用户请求处理层面	分布式系统通常通过 API 对外提供服务，用户通过这些接口向系统发送请求，如数据查询、事务处理等。然而，复杂的业务逻辑在代码实现过程中不可避免地会引入缺陷。这些缺陷可能被攻击者利用，通过构造攻击性服务请求，触发系统代码中的缺陷，从而影响分布式系统的可用性和安全性。	请求处理错误 请求超时缺陷 并发请求错误 负载均衡失效 服务降级失败 权限控制错误 … …
节点消息处理层面	在分布式环境下，节点通过消息协同，使系统作为一个统一整体运作。然而，环境复杂性和消息传递的不确定性导致节点消息处理逻辑可能存在缺陷。此外，一些分布式系统支持拜占庭容错，攻击者可伪装成普通节点发送恶意消息，利用共识代码中的缺陷，严重危害系统的安全性与可用性。	消息返回错误 消息丢失 通信 IO 错误 共识不一致 网络错误分区 并发消息死锁 … …
环境故障处理层面	几乎所有的分布式系统都支持崩溃容错（Crash fault tolerance），即使部分系统组件发生故障，系统仍能恢复正常运行。然而，分布式环境中的各种系统交互可能出现网络延迟、丢包、磁盘崩溃、CPU 错误等异常情况。不正确的故障处理、恢复代码缺陷会严重影响系统的可靠性和鲁棒性。	节点重启错误 服务恢复缺陷 部分服务故障 冗余策略失效 备份数据丢失 多故障协调错误 … …

系统配置层面缺陷：系统配置（configuration）用于定义分布式系统的拓扑结构，如服务器、网络设备、软件版本和安全策略等[31][32]，旨在提升系统的灵活性和可扩展性。然而，随着分布式系统规模和复杂性的增长，尤其在云原生等动态伸缩性系统中，配置处理变得更加复杂，增加了系统缺陷的可能性。大量研究表明，分布式系统中多数错误源于不当的配置处理[28][29][30]。例如，Barroso 和 Holzle 报告指出，配置错误处理导致约 28%的服务级故障，是谷歌主要数据中心故障的第二大原因[33]。Facebook 报告称，约 16%的服务下线事件与配置有关，其中一次配置更新错误导致 5 亿用户无法访问其网站数小时[34]。类似发现也适用于其他分布式系统，如微软 Azure、亚马逊 EC2 和谷歌等[35][36][37][38][39]。Hale 的报告显示，这些分布式系统中大部分网络故障是由于错误的配置处理代码引起的[40]。

常见的系统配置层面缺陷包括：1) 配置不一致性，即分布式系统配置项处理逻辑错误，导致实现代码与语义设计不一致；2) 配置更新失败，系统运行过程中配置更新失败，可能导致系统无法响应新的配置或配置回退错误；3) 系统配置兼容错误，不同系统配置项组合出现兼容性冲突，导致系统错误；4) 权限配置错误，权限设置不当，可能导致安全漏洞或访问控制失效；5) 依赖配置错误，配置项之间的依赖关系处理错误，导致服务无法正确启动或依赖服务不可用；6) 冗余配置项，不必要的配置项增加了代码

复杂性和错误的可能性。

用户请求层面缺陷: 用户请求（client requests）指的是用户通过服务接口向系统发送的操作请求，如数据查询、事务处理或执行特定服务的请求。它是分布式系统交互的基础，影响系统设计、资源分配和性能优化。随着业务功能的扩展，复杂的业务逻辑容易在实现过程中引入缺陷。由于大多数服务接口对外开放，用户请求层面的缺陷可能被攻击者利用，通过构造大量恶意请求触发系统缺陷，严重威胁系统的可用性和安全性。例如，2018 年 2 月，GitHub 遭遇了一次峰值达 1.35Tbps、包发送速率达 1.269 亿个/秒的 DDoS 攻击，导致服务瘫痪数小时[41]。

常见的用户请求层面缺陷有：1) 请求处理缺陷：不正确的代码实现导致请求处理过程中发生错误的行为或返回错误的处理结果。2) 请求超时缺陷：请求超时处理代码缺陷导致系统处理错误。3) 并发处理错误：多个并发请求导致的数据竞争、死锁或其他并发控制问题。4) 负载均衡失效：负载均衡机制失效，导致某些节点过载而其他节点闲置。5) 服务降级失败：在高负载或故障情况下，服务降级策略未能有效执行，导致用户体验下降。6) 权限控制错误：用户权限控制不当，导致能够访问不应有的资源或操作。

节点消息层面缺陷: 在分布式系统中，节点间协同消息（node messages）用于实现系统组件间的通信与协调，使其能统一运作并处理用户请求和内部事件[42][43]。因此，节点消息处理中的代码缺陷可能危害系统的一致性和正确性。此外，在支持拜占庭容错（Byzantine fault tolerance）的系统中，攻击者可伪装成正常节点，发送恶意消息，若代码存在缺陷，可能破坏分布式系统的共识机制，威胁数据一致性和系统安全[44][45][46]。例如，Go-Ethereum 中的共识缺陷可能导致网络分区，攻击者可利用此漏洞通过发送恶意共识消息包控制部分网络，进行双花攻击，从而破坏区块链的不可逆性并造成严重经济损失[47][48]。

常见的节点消息层面缺陷有：1) 消息返回错误：节点收到不符合预期的消息包，导致处理行为异常。2) 消息丢失：节点消息丢失后重试处理机制代码缺陷，导致协同失败。3) 通信 IO 错误：节点间通信网络 IO 处理缺陷导致消息持续阻塞，分布式协同无法进行。4) 共识不一致：攻击者发送恶意消息，利用节点共识逻辑中的缺陷破坏系统一致性。5) 网络错误分区：网络连接逻辑代码缺陷导致错误的分布式分区，影响系统一致性。6) 消息处理死锁：消息处理过程中出现死锁，导致系统停滞。

环境交互层面缺陷: 在分布式系统中，环境交互指系统与其运行环境的相互作用，包括物理环境（如硬件、网络设施）和软件环境（如操作系统、中间件）的交互[49]。良好的容灾机制能够保障系统在各种条件下稳定运行，提高适应性和鲁棒性。然而，分布式环境复杂多变，系统与环境交互中可能遇到延迟、丢包、磁盘崩溃、CPU 故障等问题，若错误处理代码存在缺陷，可能导致性能下降、数据不一致或服务中断等。例如，2017 年 Amazon Web Services (AWS) 的 S3 服务中断[50]，由一次小的网络延迟引发服务器子系统崩溃，导致大范围互联网服务中断。Facebook 平台的节点故障恢复错误[4]，导致其自身服务及使用 Facebook 身份验证的第三方网站长时间中断，造成巨大经济损失。

常见的环境交互层面缺陷包括：1) 节点重启错误：节点重启逻辑处理不当，导致分布式节点大面积瘫痪且无法恢复；2) 服务恢复缺陷：故障处理代码中的缺陷导致系统服务长时间挂起，无法处理正常业务；3) 部分服务故障：部分服务出现故障而无法响应，而非整个系统服务；4) 数据恢复失败：系统故障后的数据恢复机制不完善，导致数据丢失或无法访问；5) 冗余策略失效：冗余机制代码缺陷，导致在故障时无法提供备份支持；6) 多故障协调错误：系统在面对多故障同时发生时，未能有效协调处理，导致系统失效。

1.3 分布式系统动态测试需求与挑战

1.3.1 分布式系统测试需求

由于分布式系统具有多样的威胁输入维度和种类繁多的缺陷，对其进行测试时需要考虑的测试需求也更加复杂。根据测试目的，通常分布式系统的测试需求主要可以分为以下六大方面。

第一是功能性测试，旨在验证系统是否按预期工作，保障各模块和服务的正确执行。主要包括业务逻辑测试、节点通信测试和配置逻辑测试。业务逻辑测试检查用户请求的处理是否正确，如存储、检索和删除

操作；节点通信测试保障各节点之间的通信正常，包括消息传递、同步和回包的正确性；配置逻辑测试则验证系统的配置管理和处理逻辑是否正确，保障配置项能够被正确解析、应用和管理。

第二是安全性测试，旨在检查分布式系统的安全性并挖掘潜在漏洞。主要包括崩溃安全测试、数据保护测试和访问控制测试。崩溃安全测试检测系统是否存在内存安全、断言异常等导致系统崩溃的问题。数据保护测试验证系统的加密、敏感数据遮蔽和隐私安全存储等功能。访问控制测试检查系统的访问权限、身份验证和授权等机制。通过这些测试，保障分布式系统在各种条件下的安全性和数据保护能力。

第三是一致性测试，目的是保障系统在分布式环境下多节点或多副本之间的数据一致性和事务完整性。它包括故障一致性测试、并发一致性测试和拜占庭攻击测试。故障一致性测试验证系统在节点崩溃、网络分区或磁盘故障等情况下，能否保持数据和事务的一致性。并发一致性测试检查系统在多节点或多进程并发访问时，能否正确处理事务，避免数据竞争和死锁问题。拜占庭攻击测试验证系统在存在恶意或不可信节点时，是否能通过拜占庭容错机制维持共识一致性。通过这些测试，保障分布式系统在复杂环境中可靠运行。

第四是性能测试，旨在评估分布式系统在不同负载和压力下的响应时间、吞吐量和资源使用情况。性能测试可分为读写性能测试、并发访问测试和负载测试。读写性能测试评估读写操作的响应时间、吞吐量和资源消耗。并发访问测试检查多个用户同时访问系统时的并发性能和资源管理情况。负载测试评估系统在处理大量流量时的表现，确定系统在重载或接近预期负荷极限时的性能瓶颈。这些测试有助于在系统部署前合理配置资源，保障系统在各种负载条件下的高效运行。

第五是鲁棒性测试，目的是为了验证分布式系统在各种条件下的可用性和容错能力，包括容错和恢复测试、高可用性测试、数据备份和恢复测试以及错误处理测试等。容错和恢复测试模拟系统故障和崩溃情况，验证容错和恢复机制。高可用性测试检查系统的冗余和故障转移机制，保障高可用性。数据备份和恢复测试验证系统的备份和恢复功能，保障数据完整性和可恢复性。错误处理测试评估系统在遇到异常和错误时的处理能力和错误消息提示。这些测试保障分布式系统在各种复杂环境交互下保持稳定和高效。

第六是扩展性测试，目的是为了验证分布式系统在增加资源(如硬盘、节点等)时的表现，保障其能够有效扩展以应对增长的需求。这类测试评估系统在添加更多节点后、能否处理更多请求或存储更多数据时，是否能够保持或提高性能和稳定性。具体测试内容包括评估系统的水平扩展能力、负载分布均衡性、节点动态加入和移除的处理能力以及资源利用率。通过扩展性测试，可以识别系统在扩展过程中的潜在瓶颈，保障系统能够在各种扩展条件下高效运行，并满足未来的增长需求。

除了以上大的方面外，分布式系统还有其他的测试需求。譬如不同版本之间的兼容性测试、应对突发负载的弹性伸缩测试，以及系统从完全关闭到启动的冷启动测试等需求，都需要测试工程师针对这些需求制定相应的测试手段和开发测试工具。

因此，针对这些需求，分布式系统动态测试的总体方向是生成与待测性质相关的测试输入，并设计相应的测试准则进行检测。图 3 展示了典型的分布式系统动态测试流程，通常由一个或多个测试器协作完成。每个测试器包含两个核心组件：测试用例生成器和缺陷检测器。动态测试的总体步骤为：（1）根据系统配置启动待测分布式系统；（2）配置测试接口并启动一个或多个分布式测试器；（3）测试器生成大量测试用例，通过接口输入系统执行；（4）缺陷检测器持续监控系统行为，依据测试准则判断是否异常，生成漏洞报告。

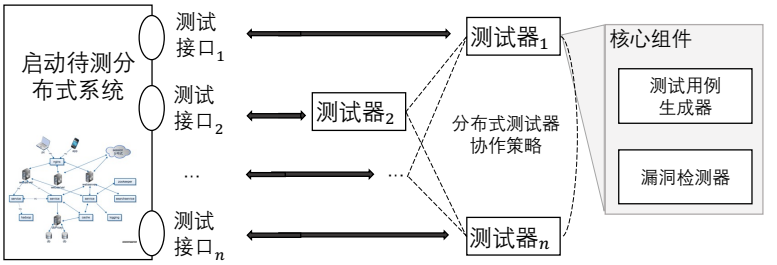


图 3　分布式系统动态测试一般架构

1.3.2 分布式系统测试挑战

结合测试需求和此动态测试框架，对分布式系统进行动态测试的挑战主要包含如下几个方面：

（1）待测系统启动设置：在分布式系统动态测试中，如何快速且准确地自动化启动待测系统是一大挑战。与传统程序相比，分布式系统环境更复杂，配置更繁琐。测试大规模分布式应用（如 GFS、AWS）时，节点数量可达 100 至 10000 个，直接在应用环境中测试可能带来不可逆损害，而模拟相同规模的环境消耗大量资源。因此，如何在有限资源下进行全面测试，降低成本并保障测试广度和深度，是关键难题。此外，测试环境选择对流程影响重大。一些工具使用物理机器搭建分布式环境，接近真实场景但成本高且维护困难。另一些工具利用 Docker 容器等虚拟化技术，灵活配置和扩展测试场景，降低资源消耗，但可能引入虚拟化偏差。还有工具将分布式系统转为单机应用进行测试，适合精确调度控制，但无法全面反映分布式特性。因此，平衡资源消耗与测试准确性、优化测试环境成为动态测试的关键挑战。

（2）分布式测试器协作： 对分布式系统进行动态测试时，有时需要部署多个测试器在分布式环境下对不同节点进行探测。在此场景下，设计和实现分布式测试器之间的高效协作策略是主要挑战之一。根据分布式测试器的协作架构，通常可以分为两种：第一种是中心化的分布式协作策略，即存在一个统筹测试器作为“中心化”协调者，负责汇总分析测试结果，并对其他测试器进行任务分配、运行调度和沟通协调。第二种是去中心化的点对点分布式协作策略，即不存在统筹测试器，每个测试器之间直接进行消息通信、任务协调和信息共享。中心化协作策略效率高，任务和数据统一处理分配，但其可扩展性较弱，需要新增缺陷检测时改动较大。点对点协作策略效率相对较低，但可扩展性高，任意新增测试器可以方便快捷地插入和拔出。因此，在分布式系统动态测试中，选择适合的协作策略至关重要，以平衡测试效率和系统扩展性。

（3）高效的测试输入生成：分布式系统测试面临多维度输入带来的语法语义多样性和组合空间庞大的挑战。与传统程序测试仅涉及单一输入不同，分布式系统的缺陷威胁来自四个维度：系统配置、用户请求、节点消息和环境交互。每个维度包含不同的语法和语义，如系统配置涵盖操作系统、网络和节点规模；用户请求涉及增删改查操作；节点消息包含握手、心跳、共识协议等；环境交互涉及网络延迟、硬盘和内存故障等。此外，输入参数的取值范围及排列组合可能影响不同执行路径。由于分布式系统处理大规模数据和高并发操作，测试需验证系统的性能和稳定性。输入空间巨大，全面覆盖所有测试组合极为困难。现有工具通常固定三个维度的输入，重点在一个维度上进行优化探索。如何协调多维度输入并高效探索测试空间，已成为分布式系统动态测试的主要挑战之一。

（4）精准的缺陷检测： 在分布式系统测试中，不同的测试需求带来不同的挑战。分布式系统的缺陷通常分为六类：功能缺陷、安全漏洞、一致性缺陷、性能问题、鲁棒性缺陷和扩展性缺陷。多样的缺陷类型使得设计精准的动态测试检测器变得复杂。目前的工具通常只针对一至两类缺陷设计检测器。例如，针对逻辑缺陷需人工编写检测逻辑，判断异常行为；针对内存安全漏洞，使用 AddressSanitizer 等工具检测内存使用；一致性缺陷通过一致性模型评估分布式行为一致性；性能问题则通过监控性能指标，结合大数据和 AI 分析异常；鲁棒性缺陷利用差分测试对比故障恢复前后的状态；扩展性缺陷则通过性能测试评估不同配置下的表现。因此，如何根据具体测试需求和系统特征设计出精准的缺陷检测器至关重要。

此外在动态测试工具的实际应用上，我们还需要面临工具可扩展性的挑战：

（5）工具扩展性：当测试工具适配于不同待测分布式系统时，需要应对可扩展性的挑战。不同分布式系统使用的架构（包括 API 规范、编程语言、网络架构、运行环境等）不尽相同。以多样的输入格式为例：分布式文件系统的输入主要是文件格式；分布式数据库的输入主要是 SQL 语句；消息中间件的输入主要是消息包。这些多样化的输入特征对分布式系统动态测试工具提出了高可扩展性的要求。为了有效测试这些不同系统，动态测试工具需要具备高度的可扩展性，以适应各种分布式架构和输入格式。目前，一些解决方案包括自动化识别主流分布式系统的 API，并人工创建输入模型和语法解析器，以适配特定系统需求。

基于这些主要挑战，我们总结出了在设计和实现分布式系统动态测试工具时，需要考虑的主要因素，如下表 2 所示：

表 2 分布式系统动态测试工具需要考虑的因素

关键功能	内容	说明
待测系统启动	节点规模	通常分布式系统应用节点规模很大，100～10000 节点不等，如何用较小的资源代价进行较为完善的测试，需要测试工具考虑。
	测试环境	测试工具需要考虑测试环境，待测分布式系统测试时运行在（1）分布式集群、（2）docker 容器、（3）单个物理机。
分布式测试器协作	测试器规模	分布式系统测试工具通常包含一个或多个输入生成器，以及一个或多个检测器。
	协作方法	在存在多个测试器的场景下，测试工具需要考虑测试器之间的分布式协作策略设计。
测试输入生成	语法语义正确性	动态测试工具生成的测试输入应保证语法和语义正确性，应当具有对自身生成的测试输入的语法和语义进行正确性验证的能力。
	用例输入的维度	测试工具应根据测试目标覆盖系统配置、客户请求、节点消息以及环境交互这四个输入维度中的各种操作。工具应支持生成具有不同操作类型和参数组合的测试输入。
	支持的运行时反馈信息	动态测试工具可以根据待测对象运行时信息反馈来提升测试效率，常见的反馈信息有代码覆盖率、关键状态覆盖率等。
缺陷检测	支持检测的缺陷类型	测试工具能够支持发现分布式系统中的各种类型的缺陷，包括但不限于功能、安全、一致性、性能、鲁棒性、扩展性缺陷等。
	缺陷判断准则	动态测试工具能够支持制定不同的缺陷判定准则来识别缺陷，如通过代码插桩、关键系统状态、实时日志信息和信号处理机制等。
	复现分析	动态测试工具应具备分析一定的缺陷复现分析能力。它应能够复现缺陷的发生过程，提取关键系统状态和日志信息，以便于进一步的对缺陷进行修复和改进。
工具扩展性	支持的分布式系统类型	动态测试工具可以支持不同的分布式架构类型，可以针对特定分布式系统进行测试，并充分利用其特性和功能发现漏洞。
	适配方法	测试工具需要能够对新的分布式系统或新版本进行适配和测试。

2 分布式系统动态测试分类和典型工具

近年来，许多分布式系统动态测试工具相继问世，发现了大量缺陷，显著提升了系统的可用性、安全性和可靠性。表 3 列出了部分典型工具及其在应对主要挑战上的支持能力。这些工具按缺陷类型分类为功能缺陷、安全缺陷、一致性缺陷、性能缺陷、鲁棒性缺陷和扩展性缺陷的测试工具。为了捕获各类缺陷，每个工具都需建立相应的缺陷判断准则。此外，测试输入合成是分布式系统动态测试的核心任务之一，不同工具针对不同输入维度设计了各自的合成技术。按照生成策略，主要分为基于规范的生成式策略和基于种子的变异式策略。许多工具依赖运行时反馈（如代码覆盖率、日志和关键状态）进行动态优化。对于测试环境的设置，工具探讨了测试时节点规模和运行环境。在应对多测试器交互的挑战时，测试工具开发了高效的协同机制。由于分布式系统架构和特征各异，大多数动态测试工具在应用时需要一定的适配工作。

2.1 针对功能缺陷的测试工具

整体介绍: 功能逻辑缺陷又称正确性缺陷,是指待测系统没有正确的实现预定义的功能。例如执行，某次数据查询或配置更新时系统返回错误的处理结果。这类问题不像崩溃问题一样会造成待测系统明显的异常，因此因此难以发现。但错误返回结果会严重影响业务逻辑处理和上层应用的执行正确性，造成潜在危害。为了检测这些问题，动态测试工具在生成测试输入时需确定预期执行结果，并通过对比实际结果和预期结果来发现问题。为实现这一目标，这类工具需要深入分析和建模待测分布式系统的输入语义，以理解其预期行为和执行规则，从而保障测试的准确性和有效性。

因此，针对功能缺陷的测试工具的主要技术难点在于制定缺陷判断准则。这需要对分布式系统的行为逻辑有深入理解，并进行详尽的分析和输入语义建模。差分测试和蜕变测试是确定预期结果的有效方法，降低了系统语义建模的难度。(1) 差分测试通过对比相似的分布式系统或同一系统的不同版本来获得预期结果。

表 3 分布式系统动态测试技术分类和典型工具

工具	年份	系统设置		测试器交互		测试输入合成			缺陷发现			工具扩展性	
		节点规模	测试环境	测试器规模[1]	协作方式	语法语义验证	输入维度	运行时反馈	缺陷类型[2]	判断准则	复现能力	支持的分布式系统	适配方法
TMT[71]	1991 (IWTCS)	–	–	1S+nT	关键事件同步	–	通信消息	无	一致性缺陷	人工定义模型	无	–	–
MODIST[103]	2009 (NSDI)	3 节点	1 物理机	1F+2M	–	满足	错误注入	无	鲁棒性缺陷	crash 信号+人工断言	手动分析	Berkeley DB, MPS, PACIFICA	人工编写适配配置文件
DieCast[72]	2011 (TOCS)	40 节点	40 虚拟机	1T	–	–	用户请求	无	扩展性缺陷	性能变化数据+人工分析	手动分析	BitTorrent, RUBiS[88],Panasas[87]	人工配置数据统计
Exalt[73]	2014 (NSDI)	>1000 节点	100 物理机	1T+1M	–	–	用户请求	无	扩展性缺陷	吞吐量收集、监控、对比	手动分析	HDFS, HBase	人工进行数据压缩
SAMC[96]	2014 (OSDI)	3 节点	1 物理机	1T+1M	–	–	错误注入	无	鲁棒性缺陷	人工定义规则	手动分析	ZooKeeper,Yarn, Cassandra	人工定义适配规则
Jepsen[91]	2015	–	虚拟机	1T+nM	–	满足	错误注入	无	功能、一致性、鲁棒性	人工定义缺陷检测规则	手动分析	分布式数据库、分布式共识协议等	人工定义适配规则
ChaosMonkey [59]	2016 (Netflix)	–	虚拟机	1T+1M	–	满足	错误注入	无	鲁棒性缺陷	人工定义缺陷检测规则	手动分析	AWS, Foundry,Azure, Kubernetes 等	人工定义故障注入点
ConfTest[76]	2017 (EASE)	1~10 节点	1 物理机	1T+1M	–	满足	系统配置	无	功能缺陷	基于日志分析	手动分析	Httpd, PostgreSQL, Yum,MySQL	人工建立配置依赖
PCTCP[89]	2018 (OOPSLA)	3 节点	1 物理机	1S+nT	–	满足	节点消息	无	一致性缺陷	基于日志分析	手动分析	Zookeeper, Cassandra	人工定义配置文件
ScaleCheck [74]	2019 (Fast)	32~512 节点	1 物理机	1T+1M	–	满足	用户请求	无	扩展性缺陷	代码插桩+关键信息提取	手动分析	HDFS, Cassandra, Riak Voldemort	手动编写测试输入
FlyMC[94]	2019 (EuroSys)	3 节点	1 物理机	1F+1M	–	满足	节点消息	无	一致性缺陷	人工定义规则	手动分析	Cassandra, Hadoop, Spark, ZooKeeper	人工编写一致性规则
CoFI[93]	2020 (ASE)	5 节点	1 物理机	1F+1M	–	满足	错误注入	一致性引导	一致性、鲁棒性缺陷	节点崩溃+异常日志	手动分析	Cassandra, HDFS, YARN	人工插桩+收集负载
Morpheus [84]	2020 (OOPSLA)	4 节点	Erlang 虚拟机	1T+1M	–	满足	用户请求	偏序关系引导	一致性、功能性缺陷	人工构建检测模型	手动分析	RabbitMQ, Mnesia, locks,gen leader	人工适配系统至 Erlang
ChaT[77]	2020 (Globecom)	56 节点	GNS3 模拟环境	nT+1M	关键消息同步	满足	系统配置	无	功能、扩展性缺陷	蜕变测试	手动分析	分布式数据传输系统	复用已有客户端应用
Ctests[22]	2020 (OSDI)	1 节点	1 物理机	1T+1M	–	满足	系统配置	异常状态引导	功能、性能缺陷	测试断言+性能对比	手动分析	HCommon,HDFS,HBase, ZooKeeper,Alluxio	复用已有集成单元测试
Frisbee[81]	2021	5 节点	5 物理机	1T+1M	–	满足	系统配置	无	功能、性能缺陷	人工定义系统配置、断言	手动分析	Redis, YCSB 等云原生系统	手动构建测试流程
TGTS[75]	2021 (IJPEDS)	20 节点	20 虚拟机	nT+1M	时序消息同步	满足	错误注入	无	性能缺陷	阈值检测	手动分析	MapReduce	手动建立时序模型
Fluffy[85]	2021 (OSDI)	2 节点	1 虚拟机	1T+1M	–	满足	客户请求	运行状态引导	功能缺陷	差分测试	手动分析	以太坊网络(geth,openethereum)	手动构建差分测试流程
DUPTester[65]	2021 (SOSP)	3 节点	1 物理机	1T+1M	–	运行时验证	客户请求	无	功能缺陷	基于历史缺陷总结规则	手动分析	Hbase, HDFS, Hive, Cassandra, YARN	手动配置映射规则
Modulo[92]	2022 (ATC)	3 节点	1 物理机	1F+1M	–	满足	错误注入	无	一致性缺陷	人工定义检测模型	手动分析	ZooKeeper, MongoDB, Redi	人工适配差异模型
MPChecker [86]	2022 (CCS)	–	1 物理机	1T+1M	–	满足	用户请求	无	权限安全缺陷	基于日志建立权限模型	手动分析	HDFS, YARN, HBase, MapReduce,Zookeepe	人工构建 API 模型
Perseus[78]	2023 (FAST)	>1000 节点	真实应用环境	1M	–	满足	客户请求	无	性能缺陷	大数据离群点分析	手动分析	分布式存储系统	自动化处理大数据
Mocket[51]	2023 (EuroSys)	5 节点	1 物理机	1T+1M	–	满足	用户请求	状态覆盖引导	功能缺陷	模型与代码实现差分测试	手动分析	ZooKeeper、Xraft、Raftjava	人工适配模型代码映射
LOKI[66]	2023 (NDSS)	10 节点	1 物理机	nT+1M (n<1/3f)	去中心化通信	满足	节点消息	共识状态引导	内存安全、功能缺陷	ASAN，交易结果建模	基于消息序列分析	区块链系统、P2P 系统	人工适配消息格式
Tyr[79]	2023 (S&P)	10 节点	1 物理机	nT+1M (n<1/3f)	去中心化通信	满足	节点消息	行为差异引导	一致性缺陷	关键信息提取+规则匹配	基于消息序列分析	区块链系统、P2P 系统	人工适配消息格式
Mallory[90]	2023 (CCS)	5-9 节点	1 物理机	nF+1M	中心化调度	满足	错误注入	时序行为引导	鲁棒性缺陷	日志分析,一致性检测	手动分析	Braft, Dqlite, Redis, MongoDB, ScyllaDB	人工插入事件批注
Phoenix[80]	2023 (CCS)	10 节点	1 物理机	nF+2M (n<1/3f)	中心化协调者	满足	错误注入	共识逻辑引导	一致性、鲁棒性缺陷	节点活性、数据一致性检测	基于序列自动复现	区块链系统、P2P 系统	人工识别共识故障点
CrashFuzz[67]	2023 (ICSE)	5 节点	Docker 虚拟机	nF+1M	中心化调度	满足	错误注入	覆盖率引导	鲁棒性缺陷	节点活性检测	手动分析	ZooKeeper, HBase, HDFS	人工识别 IO 故障注入点
ECFuzz[70]	2024 (ICSE)	20 节点	KVM 虚拟机	1T+1M	–	运行时验证	系统配置	无	功能缺陷	代码插桩+异常捕获	手动分析	Hcommon,HDFS, HBase, ZooKeeper	人工构建配置依赖
Chronos[21]	2024 (S&P)	20 节点	Docker 虚拟机	nF+1M	中心化调度	满足	错误注入	代码深度反馈	鲁棒性缺陷	节点活性检测	基于故障序列分析	HDFS,Zookeeper,MYSQL Etheruem	人工识别并插桩超时库

[1] 测试器包含全局调度器(S)、用例生成器(T)、故障注入器(F)和缺陷监视器(M)，其中 1S 代表一个全局调度器，nT 代表 n 个并行的用例生成器，1F 代表一个全局故障注入器，nM 代表 n 个并行的缺陷监视器。[2] 一些测试工具会设计多种缺陷检测器以同时检测不同类型的系统缺陷。

典型测试工具 Fluffy[85]通过持续对比以太坊分布式协议的不同语言实现版本（go-ethereum 和 opentheruem）的执行结果，成功发现了两个严重的以太坊交易执行功能缺陷。工具 Mocket[51]差分对比分布式系统设计时的形式化模型与其代码实现的逻辑不一致，成功检测出 9 个系统功能实现缺陷。(2) 蜕变测试基于已有测试执行结果，对输入进行蜕变转换以获取预期结果。典型工具 ChatT[77]通过建立蜕变关系：系统在相同负载输入、不同执行优化配置输入下，优化后的执行性能应大于等于优化前。基于此蜕变测试，ChatT 生成大量等价配置文件持续对系统的配置处理逻辑进行测试验证，挖掘出不少功能缺陷。

此外，通用特征建模和用户友好的自定义建模接口是确定预期结果和构建判断准则的常用方法。(3) 基于系统特性总结，通过分析系统设计文档、代码、用户反馈或历史缺陷，总结系统的特性或行为模式，检查实际运行时是否偏离这些特性。典型工具 LOKI[66]通过对区块链交易流程建模，构建了 liveness（合法交易最终被执行）和 safety（不合法交易不能记账）两个判断模型。在生成交易输入时，基于模型自动输出预期结果集，并在测试后对比实际结果，成功在主流区块链系统中发现了 6 个严重的交易逻辑漏洞。(4) 用户自定义模型，允许测试者根据系统逻辑理解和预期，定义一套规则或模型来检查系统行为。典型工具 Jepsen[91]允许测试人员编写 Clojure 语言脚本和配置文件，详细定义待测系统的异常行为判断条件。在测试过程中，Jepsen 实时检测这些准则并输出缺陷报告，及时识别异常行为。Jepsen 已在多种主流分布式数据库和协议中挖掘出近百个系统缺陷。

2.2 针对安全漏洞的测试工具

整体介绍：安全漏洞是指在分布式系统中发现内存安全、断言异常、错误代码实现导致系统崩溃、未授权的访问等危害系统安全性的缺陷。崩溃漏洞可以使待测分布式系统服务宕机，会严重影响其上层应用的运行，容易造成较大的损失。未授权访问漏洞陷可以导致数据泄露、敏感信息被窃取、数据篡改、系统配置被恶意修改，甚至使攻击者能够完全控制系统，造成破坏性后果。

崩溃漏洞检测是在待测系统执行测试输入过程中，测试工具会监控系统的分布式进程状态，判断是否出现崩溃信号。考虑到内存安全问题（如缓冲区溢出、栈溢出）在早期不会出现明显症状，测试工具常使用 AddressSanitizer(ASAN) [95]对待测代码进行插桩处理，实时监控内存异常访问并提前抛出崩溃信号。由于检测崩溃漏洞是动态测试的基础功能，所以这一类工具的主要技术难点主要集中在生成高质量的测试用例输入以高效触发系统崩溃。典型测试工具 LOKI[66]通过实时感知分布式系统的运行状态，自动构建关键消息状态机模型，并基于此优化测试消息输入，提高测试覆盖率，检测更多安全漏洞。LOKI 认为消息状态机覆盖反映了测试工具对分布式系统测试的完备程度。如果一个消息测试输入可以触发更多的新的状态、状态转移，LOKI 将其视为一个有意义的测试输入，可以通过对其进行变异来触发之前未覆盖的消息状态转移。LOKI 通过收集每个消息测试输入的状态转移数，引导消息测试输入的变异，从而提高生成质量。在对 Go-Ethereum、Fabric、FISCO BCOS、Diem 等系统的 24 小时测试中，LOKI 发现了 14 个崩溃漏洞。

未授权访问漏洞检测是指通过分析系统行为和权限检查逻辑，识别系统在执行特权操作时未进行适当权限验证的情况。MPChecker[86] 首先通过静态分析对系统代码进行扫描，识别出所有可能涉及权限检查的代码路径和特权操作点。对这些操作点建立权限检查模型，记录系统在执行这些操作前应进行的权限检查逻辑。基于权限检查模型，工具自动生成一系列测试用例，这些测试用例模拟各种合法和非法的权限操作。这些用例旨在触发系统的各种功能，并测试系统对不同权限请求的响应。如果有违反权限模型的访问操作，这被视为未授权访问漏洞。Morpheus 在 HDFS、HBASE、YARN 等主流分布式系统上进行了长时间的测试，并最终挖掘了 44 个新的未授权访问漏洞。

2.3 针对一致性缺陷的测试工具

整体介绍：一致性缺陷是指在分布式系统中，由于共识同步不正确、并发处理不当或故障处理错误等原因，导致系统中多个节点或副本之间的数据不一致、事务无法正确完成等缺陷。这类漏洞会造成数据的不完整性和不准确性，严重影响系统的可靠性和上层应用的正常运行。分布式系统的一致性缺陷主要包括共识不

一致缺陷、并发冲突一致性缺陷、崩溃一致性缺陷等。这些一致性缺陷可能导致数据丢失、系统操作失败，甚至业务逻辑错误，从而引发重大经济损失和信誉风险。

共识不一致检测旨在分析和验证分布式系统中多个节点在决策过程中的一致性，以保障数据一致性和系统的正确运行。针对去中心化分布式系统的共识一致性问题，典型测试工具 Tyr[79]设计了一种基于行为分化导向模型的模糊测试算法。在分布式环境中，实时一致性难以实现，多数系统仅保证最终一致性。Tyr 认为短暂的不一致是正常且可接受的，但最终不一致通常是由多次短暂不一致积累而来。因此，Tyr 将导致节点行为差异的消息测试输入视为有价值的测试输入，并通过变异扩大这种差异性。Tyr 首先对去中心化分布式系统的共识过程进行行为差异建模，并在测试中实时收集计算节点间的行为差异，进而引导消息测试输入的变异，提升生成质量。在对 Go-Ethereum、Fabric、FISCO BCOS 和 Diem 等常见共识系统的 24 小时测试中，Tyr 发现了 20 个严重的共识不一致缺陷。

并发冲突一致性检测通过分析和验证分布式系统中多个并发操作的相互影响，确定是否会导致数据不一致和状态异常（如数据损坏、死锁等）。其最大挑战是有效探索和覆盖庞大的并发操作空间。由于分布式系统中并发操作数量众多，操作之间的交互复杂且不可预测，测试时难以穷尽所有可能的并发场景。典型工具 PCTCP [89] 利用随机调度算法，将消息交互建模为事件，建立有序事件集，并在运行时动态选择和执行事件，提高发现并发错误的概率，在主流分布式系统中发现了 9 个新缺陷。但由于事件有序关系有限，大多数事件依赖关系不确定，测试探索空间依然很大。针对这一问题，典型工具 FlyMC [94] 通过识别系统状态对称性和事件独立性，减少状态和事件组合数量，并并行化处理多个独立事件来加快测试过程。此外，FlyMC 通过并行翻转多个事件顺序，系统化地探索复杂事件交错，从而提高检测效率，在主流分布式系统中发现了 10 个新一致性缺陷。然而，系统性地探索整个输入空间的测试效率依然较低。因此，典型工具 Morpheus [84] 仅针对复杂并发执行路径，利用部分顺序采样技术，专注于探索稀有但关键的并发冲突。在测试过程中，Morpheus 实时分析系统运行状态，动态调整事件优先级，根据关键事件进行调度，提高发现并发冲突的概率，在主流分布式系统中发现了 11 个严重一致性缺陷。

崩溃一致性缺陷检测是指在分布式系统中，检测由于节点崩溃、系统故障等原因导致的系统数据不一致和事务失败的缺陷。该检测主要关注系统在故障后，是否能够正确恢复数据一致性状态、完成未决事务以及防止数据丢失或重复。典型工具 Modulo[92]对分布式系统的操作事件、故障事件、分歧状态、恢复事件以及收敛路径（从分歧状态到一致状态的收敛路径）进行建模，生成分化同步模型 DRMs。基于该模型，Modulo 生成事件调度计划，包括一系列系统操作和故障注入事件，引导系统进入特定的分歧状态。系统执行这些计划时，Modulo 实时比较不同节点的数据状态，验证系统在故障恢复后的收敛情况，保障系统数据能正确达到一致状态。Modulo 在 ZooKeeper、MongoDB 和 Redis 等主流分布式系统发现了 6 个新的崩溃一致性缺陷。另一个典型工具 Phoenix[80]通过在区块链系统中随机注入上下文敏感的故障，在区块同步、共识投票等关键步骤引入一系列故障注入技术，如磁盘数据故障、数据污染等，来模拟可能导致系统数据不一致的场景，在 Ethereum、fabric 等主流区块链系统上发现了 6 个新的崩溃一致性缺陷。

2.4 针对性能问题的测试工具

整体介绍: 性能问题通常指目标分布式系统不能在给定的时间范围内返回请求处理结果。分布式系统通常用于处理大量并发请求和大规模数据，如果性能不佳，系统可能无法及时响应用户请求，影响用户体验，导致服务质量下降。此外，性能问题还会增加资源消耗，导致运营成本上升。相比于其他缺陷类型，性能问题的确定具有更多的不确定性。崩溃问题会对系统造成明显的影响，逻辑缺陷和一致性缺陷有清晰的判断标准，而合理的响应时间则难以给出一个明确的界限。因此，针对分布式系统性能问题的测试工具重点是设计合理的缺陷判定准则。目前大多数测试工具采用的是基于数据分析的离群点识别算法来判断系统的性能变化。

以微服务架构为典型特征的云原生系统具备高度的动态伸缩性，其配置管理、服务实例动态变化等提升系统性能的机制主要由 Kubernetes 等平台层提供支持。针对云原生系统的性能问题检测，典型工具 Frisbee[81]提出了一个综合框架，通过声明式方法，允许用户定义系统配置和测试工作流程，并在 Kubernetes 环境中自

动化执行这些配置、动态加载请求、注入网络故障如延迟等，监控和验证系统性能。虽然用户自定义的判断标准是有效的方法，但通常要求用户对待测系统有深入了解，且需要投入较高的人力成本。

此外，分布式系统在开放网络中长时间运行过程中，某些硬件或软件组件可能会因各种原因导致性能大幅下降，尽管它们仍能提供服务，但整体性能受影响。IASO [97] 针对这类性能问题，设计了基于超时信号的检测机制。IASO 构建了一个超时评分模型，实时监控并计算每个节点的超时得分，生成一个超时分数集。随后使用 DBSCAN [55] 算法聚类这些得分，识别并排除噪声点，将噪声点视为性能缺陷点。IASO 在一个包含 39,000 个节点的集群中成功捕捉到 232 个性能问题。然而，IASO 未考虑真实应用场景中的网络延迟和负载变化，导致许多误报。为了解决这一问题，Perseus [78] 使用轻量级回归模型快速定位和分析存储中的性能问题。Perseus 收集每个节点的实时数据，如磁盘吞吐量和写延迟，并建立一个系统负载和正常磁盘写延迟的回归模型。系统运行时，实时检测存储磁盘的写延迟，若不符合回归模型，则视为离群点，即性能问题。通过该方法，Perseus 在阿里云上找到了 315 个性能问题，这些问题修复后极大的增强的阿里云的可用性。

2.5 针对鲁棒性缺陷的测试工具

整体介绍: 鲁棒性缺陷是指分布式系统在面对异常情况（如硬件故障、软件错误、资源耗尽等）时无法稳定运行或正确处理错误的缺陷，影响系统的可用性、容错能力和稳定性。由于分布式环境中故障种类多样（如硬件、网络、资源耗尽）且故障发生时机不确定，故障类型及其排列组合的输入空间巨大。因此，测试鲁棒性缺陷的主要难点在于设计高效的故障输入生成算法，以覆盖多样的故障场景。

早期研究工作关注实现级别模型检测技术，将模型检测应用于分布式系统动态测试中，通过捕捉系统行为的深层语义进行建模，并模拟故障场景以检测鲁棒性缺陷。典型工具如 MODIST[103]和 SAMC[96]通过建模分布式系统的主要操作（如增、删、改、查等）和故障操作（如网络延迟、磁盘故障、内存故障等），抽象关键状态（如数据存储、复制、同步），形成状态转换模型，并定义关键属性（如数据不丢失、节点故障后恢复）。这些工具在系统源码中插入监控逻辑，记录状态变化和操作事件，定义测试场景模型，并基于模型自动生成测试用例，确保覆盖各种状态转换和边界情况。在执行测试时，工具实时比较实际运行状态和模型预期行为的差异，验证关键属性是否满足。如果存在差异或关键属性未被满足，工具将检测出鲁棒性缺陷。通过这种方式，模型检测技术为分布式系统的鲁棒性测试提供了有效支持。

然而，对分布式系统进行建模需要很强的领域知识，并会花费较大的人力成本，同时常常会面临模型状态空间爆炸问题，复杂系统可能难以完全建模和验证。为了解决这一问题，后续的故障注入工具如 ChaosMonkey[59]和 Jepsen[91]等，不采用建立模型的方法，而是直接在测试环境中随机注入故障。ChaosMonkey 在待测系统运行期间随机终止节点或服务，观察系统如何处理这些故障并恢复正常运行，帮助开发团队验证系统在部分组件失效时的响应能力。此外，Jepsen[91]还会随机模拟网络分区、时钟偏差以及各种软硬件故障，测试分布式系统的容错能力。除随机注入测试外，Jepsen 还提供了一系列用户友好的测试接口，允许测试人员通过编写 Clojure 语言的代码脚本及相应配置文件，详细指定故障注入策略，如故障类型、注入时机、持续时长及其影响范围，从而增加了工具的灵活性。随机探索的方法虽然放弃了验证的完备性，但极大地提高了工具的可用性，并在众多分布式系统中发现了上百个鲁棒性缺陷。

考虑到随机探索容易遗漏许多故障输入空间，模糊测试技术擅长探索输入空间并触发系统缺陷，因此多项研究尝试将模糊测试与故障注入结合，以挖掘分布式系统的鲁棒性缺陷。典型工具 CrashFuzz[67]利用代码覆盖率作为引导信息，在测试中动态优化故障序列，相比随机探索测试提高了 20%以上的代码覆盖率，发现了 4 个新的鲁棒性缺陷。Mallory[90]在测试过程中动态捕捉关键事件的顺序，建立事件“先发生关系”（happen-before），并使用 Q-learning 优化故障序列，以最大化系统行为的观察，增加触发缺陷的可能性，发现了 22 个新的鲁棒性缺陷。针对分布式系统的超时逻辑，Chronos[21]认为深层路径的超时交互逻辑很少在常规测试中被触发，可能隐藏许多缺陷。Chronos 设计了深度优先的模糊测试算法，实时计算每个故障序列触发的超时逻辑深度，优先变异可达深层超时逻辑路径的输入，成功在主流分布式系统中发现了 27 个新的鲁棒性缺陷。

2.6 针对扩展性缺陷的测试工具

整体介绍:可扩展性缺陷是指分布式系统在增加资源（如增加节点或存储容量）时，无法保持或提升其性能和稳定性的缺陷。这类缺陷会影响系统在扩展过程中的表现，限制上层分布式应用的业务扩张发展。由于真实分布式应用通常需要成百上千个分布式节点来运行，而在测试过程中通常难以获取如此多的计算资源来模拟和测试系统的可扩展性。因此，针对可扩展性缺陷的测试工具的主要难点在于，如何在有限资源下，探索有效进行大规模节点的模拟和测试的方法。

为了解决分布式系统扩展性测试中的资源限制问题，Gupta D.等人于 2011 年提出了 DieCast[72]测试工具。DieCast 通过在少量物理机器上多路复用分布式节点的虚拟机（VM），并精确调整 CPU、网络和磁盘资源，使每个 VM 在计算资源和通信行为上都能模拟原始服务机器。这种方法使 DieCast 能够在较少物理资源的情况下，有效模拟原始服务的行为。针对大规模分布式存储系统的扩展性测试挑战，2014 年 Wang Y.及其团队开发了 Exalt[73]测试库。Exalt 引入了一种创新的数据表示方式，使得在底层存储层面上也能实现数据的有效识别和压缩，即便这些层次不理解更高层次的语义和格式。这一技术极大地促进了分布式节点在物理资源上的高度共存，使得在有限机器上进行数以万计节点的大规模实验成为可能。工具 ScaleCheck[74]是一种针对大规模数据处理系统可扩展性缺陷的测试工具，它能够在单台物理机上部署并测试整个分布式系统。ScaleCheck 利用全局事件驱动架构（GEDA）在单进程集群中模拟分布式节点间的通信，并通过处理幻觉（PIL）技术进行非侵入式修改，以有效管理和协调机器上数百个 CPU 密集型节点的资源使用。这使 ScaleCheck 在使用较少计算资源的情况下，成功检测如 HDFS 和 Cassandra 等真实系统中的扩展性缺陷，累计发现 10 余个严重缺陷。类似的工具 Minha[82] 是一个专为 Java 语言实现的分布式系统测试而设计的工具，它通过在单个 JVM 中虚拟出多个 JVM 实例来模拟分布式环境。这种设置允许每个虚拟节点仿佛运行在独立的机器上，拥有自己的网络和 CPU 资源。框架支持并发和分布式 JVM 字节码程序的运行，可扩展至数千个虚拟节点，并支持对整个系统的实时全局观察。

2.7 测试环境设置与测试器协作策略

在对分布式系统进行测试时，除了关注检测目标缺陷类型外，一些测试工具通常会关注待测分布式系统的测试环境设置问题，并使用测试器同时对接不同分布式节点，以更精准及时地获取系统的关键状态，提高测试效率。然而，多测试器的设计会引入分布式测试器之间的高效协作难题。

早在 1991 年，Andreas W. 等人提出了分布式系统动态测试架构 TMT[71]，该框架包含多个分布式测试组件，用于与待测系统交互并生成测试输入；一个全局测试者则负责观察系统内部的交互，并通过同步事件协调并行测试器。随后，Hsaini S. 等人[75]提出基于时序信息同步的分布式测试协调策略，开发了一种生成本地定时测试序列的新算法来描述每个端口的行为。此算法有效减少了测试器间的消息交换量，提高了测试效率。然而，该策略仅限于时序逻辑缺陷检测，无法支持新的缺陷检测类型。为应对去中心化分布式系统测试中的协作难题，LOKI[66]和 Tyr[79]提出了点对点协调通信模式的策略，使测试节点伪装成正常服务节点，利用现有通信机制进行测试器间协调。此方法旨在减少测试工具的适配成本并提高可扩展性，但在通信效率上不及中心化协调模式，尤其是在需要实时控制分布式系统时。因此，针对精准故障注入测试，Phoenix[80]和 Chronos[21]等工具采用了中心化协调者设计，实时监控分布式节点的运行状态，并动态优化测试流程。中心化设计能够更好地收集关键信息，以便在整个测试过程中作出及时调整。此外，由于云原生环境下的弹性、动态资源分配和容器化等特性，使得动态测试更加高效和灵活。通过容器化技术（如 Docker 和 Kubernetes），能够快速构建、协同和调整分布式测试器，大大降低了传统静态测试环境的开销。典型工具 Frisbee[81]、CND[61]、ChaosBlade[56]等提出了一个接近生产环境的测试方法（Preproduction Deploys），即在部署至生产环境之前，通过统一的容器管理接口协同不同分布式测试器，将微服务或组件在一个仿真度非常高的测试环境中运行，模拟实际的工作负载和流量。这种方法能够更好地捕捉分布式系统中的复杂错误，尤其是难以在本地测试环境中发现的集成问题。

3 分布式系统动态测试的关键技术

本章节介绍了分布式系统动态测试的三项关键技术：测试输入生成、系统状态感知和缺陷判断准则构建。这些技术在动态测试中相互关联、协同工作，提升测试工具的效率和准确性。根据图 3 的分布式系统动态测试框架，测试工具面临输入语法语义正确性、输入空间高效探索、测试准则制定和缺陷判定等挑战。这三项技术共同应对这些挑战：测试输入生成技术提供多样化的输入，帮助测试工具覆盖更多情况；缺陷判定准则构建技术帮助识别和判定测试结果的合法性；系统状态感知技术则起承上启下的作用，通过监测系统状态，一方面优化测试输入的探索策略，另一方面为判定准则提供实时数据。这些技术的结合可有效提高动态测试工具的效率，帮助发现分布式系统中的潜在缺陷，提高系统的稳定性和可靠性。

3.1 测试输入合成技术

测试输入生成技术是动态测试的基石，旨在生成覆盖丰富系统逻辑的测试输入，以尽可能多地覆盖目标代码并触发潜在缺陷。与传统应用程序相比，分布式系统的测试入口维度更多，场景更加复杂。基于图 2 的分布式系统运行流程，测试输入生成通常需关注四个主要维度：系统配置生成、用户请求负载（workload）合成、节点消息构造以及环境错误注入。

3.1.1 系统配置生成技术

在分布式系统测试中，将系统配置作为测试输入可以全面评估系统在不同配置下的行为和性能，揭示配置相关的逻辑错误和性能问题，保障系统在各种环境中稳定运行。系统配置生成技术的核心在于根据系统配置项的语法规则和语义约束，生成具有复杂性和多样性的配置文件，以触发系统在不同网络架构和运行环境下的异常行为（如崩溃、资源泄露等）和处理逻辑错误（如性能优化失败、配置加载错误等）。由于配置文件结构复杂，对语法和语义要求严格，生成合法且有效的高质量配置文件成为测试工具的关键挑战。如下图 4 树状模型所示，一个分布式系统配置通常包含两个主要部分：通用配置部分和特定系统配置部分。在通用配置部分下，有节点、网络规模、CPU 使用、内存使用和硬件资源等子分支。在特定系统配置部分下，有数据存储格式、查询优化选项和负载均衡策略等子分支。目前，根据配置输入执行模式的不同，系统配置生成工具主要分为：系统静态配置生成技术和系统配置动态（on-the-fly）更新技术。

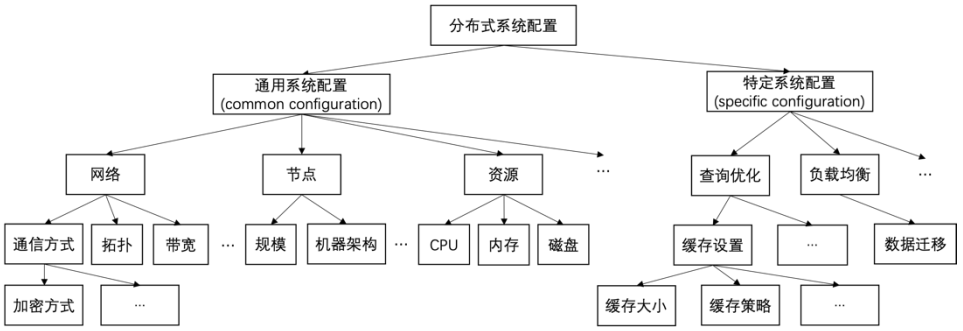


图 4 分布式系统一般配置模型

系统静态配置生成：测试工具通过基于系统配置项生成配置文件作为测试输入，并循环加载这些配置文件来启动待测分布式系统进行测试。首先，根据待测系统的配置语法和语义预先定义生成规则，然后依照这些规则生成大量配置文件。为了保障生成的配置文件语法和语义正确且多样化，通常会建立一个配置输入模型，该模型概括了关键配置选项的语法结构、语义依赖和取值范围等信息。此外，为了提高测试的全面性和系统覆盖率，还会对生成的配置文件进行探索，通过变更配置项的组合生成新的配置版本，从而更全面地探索系统的不同执行路径和潜在缺陷，增强测试的深度和广度。

针对分布式系统的通用配置输入，典型测试工具如 TEA-Cloud[83]和 ChatT[77]等，专注于变更待测系统

的计算资源使用情况，包括物理机数量、CPU 使用数量、网络带宽和磁盘存储量等。TEA-Cloud 通过定义一套形式化的生成规则，创建涵盖不同计算资源配置的配置文件。工具会系统地生成大量配置文件，并在不同资源配置下反复启动和测试分布式系统，评估其性能变化。具体来说，TEA-Cloud 通过变更计算资源和网络资源的组合，生成不同版本的配置文件，从而模拟各种资源配置场景，检测系统在这些场景下的表现。ChatT 通过变异测试技术生成系统配置作为测试输入。ChatT 首先建立一个基本的配置文件，然后通过系统地变异关键配置项（如 CPU 分配、内存使用、网络带宽等），生成一系列变异配置文件。工具通过这些配置文件加载系统，评估在不同资源配置下的系统性能表现。ChatT 的重点在于通过比较系统在不同配置下的性能，检测关键配置项的变更是否会导致系统性能的显著下降或异常行为。

针对分布式系统中特定业务逻辑相关的配置测试输入生成问题，典型工具包括 ConfTest[76]和 ECFuzz[70]。ConfTest 通过分析系统的配置选项（如权限、负载、数据配置等）、语法结构和语义约束，建立详细的配置模型，并基于此生成涵盖各种配置情况的配置文件，包括正确配置和故意引入错误的配置文件。ConfTest 通过变更配置项，生成多种配置组合，并利用系统自带的单元测试加载这些配置文件，检测配置相关的逻辑错误。对于大型分布式系统，由于配置参数众多，配置输入组合空间非常庞大，导致测试效率低下。为解决此问题，ECFuzz 采取了多维配置生成策略。该策略首先基于不同参数间的依赖关系制定多样化的变异策略，然后在每轮测试输入生成过程中，从备选的配置参数集合中挑选多个参数进行组合，以此有效降低状态空间的探索范围，并生成有意义的配置测试输入。鉴于许多生成的配置测试输入在语义上相似且不太可能引入新的错误，ECFuzz 为了提升测试效率，会先运行分布式系统已有的单元测试输入。此步骤旨在过滤掉那些不太可能引发错误的配置参数，从而高效验证新生成的配置参数的有效性，优化了测试过程。

系统配置 On-the-fly 更新：考虑到提高系统的灵活性和可用性，大多数分布式系统设计了在运行时动态修改配置选项的能力。这样的系统配置动态更新机制允许测试工具在分布式系统运行测试时，根据实时的系统状态和性能反馈，灵活地在线变更配置，从而更高效地探索测试的状态空间，揭露系统中更深层次的代码缺陷，提升测试的有效性和系统的整体安全性。

典型配置更新测试工具 Ctests[22]通过变异分布式系统自带的测试流程，在线检测分布式系统的配置更新逻辑。具体来说，工具首先采集当前系统的配置文件，包括所有配置项的当前设置值，并记录系统的初始状态和性能指标作为基准，用于后续的配置变更效果对比。随后，Ctests 工具通过变异技术生成多个不同的配置组合测试输入，待测系统 on-the-fly 实时更新这些系统配置。Ctests 持续监控系统的关键性能指标，对比初始状态，识别异常行为、性能瓶颈和系统状态信息，形成实时反馈数据。基于反馈数据，工具动态优化选择需要调整的配置项，生成新的配置更新输入。针对分布式数据库系统的配置动态更新逻辑，典型工具有 PARACHUTE[98]。它的关键洞察在于，对于任意一个配置选项，无论其值是在系统启动时被加载还是在系统运行中被更新，其对目标系统的影响应保持一致。基于这一理念，PARACHUTE 通过持续变更已有的系统配置选项，动态生成新的配置更新测试输入，并通过比较系统启动加载与运行时更新配置后的执行结果来检测结果的一致性。若发现结果不一致，则可能指向系统配置的潜在缺陷。

3.1.2 用户请求负载合成

在分布式系统测试中，用户请求负载合成通过公共服务接口模拟用户行为，生成测试输入。一方面，这些输入用于模拟不同规模的并发请求，评估系统在高负载下的性能和稳定性，识别性能瓶颈并为优化提供依据。另一方面，输入需具备“攻击性”，模拟潜在攻击场景，测试系统的安全防护能力，揭示如输入验证不足、权限配置错误等安全缺陷，帮助提升系统安全性。面对分布式系统功能多样的挑战，生成全面覆盖所有功能的测试输入是主要难点。当前，用户请求负载合成主要有三种方法：模型驱动的负载生成、API 合成和系统测试用例转换技术。

模型驱动的负载生成（Model-Driven Workload Generation）：通过系统模型自动生成测试负载。该方法通过理解系统的行为模式和性能特征，构建模型来模拟实际用户或系统活动。首先，测试人员需对分布式系统的架构、功能和用户行为进行深入分析，构建反映关键行为和性能特征的模型。随后，基于该模型，测

试工具可自动生成大量负载测试输入，用于评估系统性能和稳定性。

Bodnarchuk R 等人于 1991 年首次提出了用于分布式文件系统测试的负载生成模型 SUE[60]。SUE 首先收集分布式系统在真实环境中的用户数据，进行特征工程，构建负载特征描述模型，识别对系统性能影响最大的工作负载。基于这些特征，SUE 合成驱动程序生成负载，持续对系统进行测试。使用实际数据集作为负载生成基础是一种简单有效的方法，能够更真实地反映实际使用情况，但需要对用户隐私数据进行清洗处理。通过研究分布式工作负载，研究者发现其具有两个关键特征：突发性和自相似性。突发性指的是在短时间内数据包或用户请求剧烈波动；自相似性则指长时间内呈现相似的周期性波动。针对这两个特征，典型负载生成工具 BURSE[58]基于两态马尔可夫调制泊松过程（MMPP2s）叠加模型来模拟这些特征。该模型简单易用，所需的输入参数可直接从系统日志导出或由性能分析人员提供。基于此模型，BURSE 能生成大量具有突发性和自相似特征的负载，用于分布式系统部署前的模拟测试，有助于评估系统的稳定性与性能。

基于 API 的负载合成技术：手动构建测试模型不仅耗时耗力且缺乏灵活性，特别是面对不同分布式系统时，为每个系统定制的负载生成模型往往无法通用，限制了测试方法的普遍适用性。鉴于分布式系统通常提供详尽的 API 文档来描述其对外服务和接口，这为自动化生成负载测试输入提供了可能。基于 API 的负载合成通过利用系统提供的服务接口（API）来创建大量测试负载。该方法的核心在于使用 API 调用序列模拟用户操作或系统交互，以此来测试并评估分布式系统在处理各种用户请求负载时的性能和稳定性。为了提高测试效率，一些动态测试工具采用实时监控技术来获取系统运行的关键信息，并据此实时调整 API 调用序列的生成策略，从而有效提升探索系统潜在问题的能力，深入挖掘特定功能和执行路径的相关缺陷。

典型工具如 Swagger(先称为 OpenAPI)[57]提供的自动化代码生成工具可以用来创建 API 的模拟实现（Mock Servers），也可以从 API 描述文档自动生成测试输入，允许测试人员在不访问实际后端服务的情况下测试 API。JMeter[29]则是另一个广泛使用的开源负载测试工具，它通过待测系统的 API 描述，模拟多用户并发访问来测试分布式应用和服务，包括但不限于 HTTP/HTTPS、SOAP/REST 等服务。使用 JMeter 进行分布式系统测试时，测试工程师首先会创建一个测试计划，其中包含了一系列 API 调用请求，这些请求模拟了用户对系统的操作。然后，JMeter 可以在单机上模拟数百到数千个并发用户生成负载，也可以通过配置多台机器的 JMeter 实例形成一个测试集群，进一步扩展测试的规模。在测试执行过程中，JMeter 收集各种性能指标，如响应时间、吞吐量和错误率，帮助开发者识别系统瓶颈和性能问题。针对分布式微架构服务，工具 Pact[28] 设计了一个消费者驱动的契约测试模式，专为测试分布式系统中的服务间的 API 交互。它通过在服务的消费者（客户端）和提供者（服务器端）之间建立契约：首先是由消费者定义对服务提供者的预期行为，这些预期形成了一份“契约”。这份契约包含了每个 API 预期的请求和对应的响应。然后，这份契约会被传递给服务提供者，服务提供者利用契约进行自己的测试，保障其实际行为符合契约规定的预期。这种方式使得每一方都在确的预期下进行开发和测试，从而减少了服务间集成时的不确定性和错误。

系统测例转换技术：考虑到大多数分布式系统在开发过程中都会编写大量的系统测试用例（包括集成测试和单元测试用例），用于验证和评估其功能是否符合设计要求。因此，一些动态测试工具， Ctests[22]、DUPTester[65]等，会基于系统自带的测试用例进行分析，自动转换生成测试用例合成器。以 DUPTester 工具为例，它通过将原本的单元测试输入转换为客户端 Python 程序，生成大量用于系统测试的用户请求负载输入。具体来说，DUPTester 首先为每个单元测试构建一个抽象语法树（AST），然后基于这个 AST 合成一个 Python 程序。在这个过程中，DUPTester 需要处理直接翻译不能工作的情况。例如，DUPTester 将一些 Java 库函数和类替换为相应的 Python 函数和类，如将 Java 的 HashMap 对象替换为 Python 字典对象。此外，DUPTester 将一些特定于软件的测试方法或软件内部函数替换为可以从客户端发出的相应 Python 函数。例如，对于 Cassandra，DUPTester 用对 Python 程序中 Cluster.Session.execute(q)的调用来替换单元测试方法 CQLTester.execute(q)的调用。然而，考虑到业务逻辑和代码语法的复杂性，DUPTester 不能保证正确翻译所有语句。因此，DUPTester 提供了一个接口，允许测试人员指定单元测试方法或软件内部函数（在单元测试中调用）与外部命令（可由客户端发出）之间的映射关系，并基于该映射关系完成测试用例的转换。

3.1.3 节点消息构造技术

节点消息构造技术基于分布式基础通信架构，模拟生成节点交互通信包，生成一系列通信消息测试输入。一方面，通过构造分布式环境中各节点间交互通信的测试输入，可以模拟各种可能的正常和异常节点间消息交换场景，以验证这些协议和接口的实现是否符合规范，保障系统组件之间能够正确交互，同时评估系统在处理这些交互时的性能、稳定性和安全性。另一方面，通过构造异常或恶意设计的节点间消息，测试输入可以揭示分布式系统中潜在的安全缺陷，如信息泄露、拒绝服务攻击（DoS）、拜占庭攻击等。考虑到节点间通信消息往往是高度结构化、语义化的，生成高质量且语法语义正确的消息是消息生成工具的主要探索目标。目前，典型节点消息构造技术主要有两大类：基于模型的消息生成技术和基于种子的消息变异技术。

基于模型的消息生成技术：在分布式系统动态测试中，基于模型的消息生成技术指的是利用预定义的模型来自动生成用于测试的消息通信包。模型通常包括包括状态模型和数据模型。状态模型描述了系统或组件可能处于的各种状态以及在接收到特定消息时如何从一个状态转换到另一个状态；数据模型则定义了消息的结构、格式和有效值范围，用以生成符合实际通信协议或业务逻辑的测试数据。

Michael D 等人设计的测试工具 netFuzz[64] 通过生成大量消息通信包对分布式系统进行模糊测试。首先，它提供了一个用于描述消息的模板，用户可以基于该模板自定义待测系统的消息语法格式和关键语义字段等。随后，netFuzz 将该模板信息自动解析转换为状态模型和数据模型，并基于这些模型生成大量的消息测试包，通过网络插件接口发送给待测系统。此外，Peach[54] 是一个针对各种网络协议进行模糊测试的常用工具。测试者编写包含待测目标数据模型和状态模型的 pit 文件，Peach 根据这些模型自动生成大量通信测试包。Peach 使用数据模型来定义协议消息的结构和行为，描述节点之间交换消息的格式。Peach 还使用状态模型来定义消息交换的顺序，捕获协议的逻辑。一旦定义好这些模型，Peach 会系统地对消息和序列进行变异，生成各种测试输入，并将这些输入发送到分布式系统中。通过监控系统对这些输入的响应，Peach 帮助发现分布式协议中的缺陷、不正确的实现和潜在的安全问题。

基于种子的消息变异技术：基于种子的消息变异技术是通过对已知有效的消息样本（种子）进行系统性修改和扰动来生成新的测试消息的方法。该技术旨在通过轻微改变种子消息的结构或内容，探索待测系统对异常或边缘情况输入的处理能力，从而揭露潜在的错误或缺陷。这种方法通常应用于灰盒测试流程中，通过分析待测系统的运行时反馈信息，动态调整变异策略，以提升测试效率。

典型动态测试工具 LOKI [66] 和 Tyr [79] 均使用基于种子的消息变异技术来构造拜占庭攻击包，以检测分布式共识协议中的缺陷。具体来说，在测试启动阶段，它们将自己伪装成正常节点，接入区块链网络。随后，它们实时监控网络中的消息通信包，收集正常节点间的消息包作为初始种子加入种子池中。由于分布式系统中的消息通信包种类繁多且高度结构化，消息变异时必须保持其语法结构的正确性。因此，LOKI 和 Tyr 在变异消息时，仅变异每个包字段的内容，而不破坏其结构。对于不同类型的字段，它们使用不同的突变操作：数值突变器随机将数值类型转换为另一个数字，字符串突变器修改字符串生成新字符串，结构体对象类型则递归地突变结构体中的每个变量字段。在测试阶段，为了尽可能探索更多共识状态，LOKI 实时收集待测系统运行时的状态转移信息，并以此作为反馈引导消息的变异过程。任何能导致新状态转移的消息包被认作有趣的消息种子，保留进种子池用于后续测试变异。而 Tyr 为了更频繁地触发分布式节点共识不一致缺陷，实时收集分布式节点之间的行为信息，并基于行为分化模型计算它们之间的行为差异。任何能导致节点行为差异增加的消息包被认作有趣的消息种子，保留进种子池。通过这些方法，Tyr 能持续分化分布式节点的行为逻辑，更快更多地挖掘共识不一致缺陷。

并行消息调度技术：此外，针对分布式系统的消息并发处理问题，除了保障通信消息的语法和语义正确性外，还需要关注分布式环境下的并发消息执行顺序问题。由于分布式系统中消息交互顺序空间非常庞大，不同种类的消息之间存在各种排列组合，如何对消息序列进行高效探索成为研究者面临的主要挑战之一。

因此许多工具针对分布式节点间、甚至更细粒度的进程间通信消息调度执行顺序问题，提出了各自的输入空间优化算法，以高效触发系统中潜藏的并发不一致缺陷、行为状态异常等问题。典型工具 PCTCP [89]

利用随机调度算法，将消息交互建模为事件，建立执行顺序事件集，并在运行时动态选择和执行事件，从而提高发现并发错误的概率。具体而言，PCTCP 在每个调度点随机选择下一步要执行的事件，这种随机调度直接忽略了输入空间的庞大问题，有助于探索不同的事件交互顺序。

考虑到分布式系统中事件顺序关系挖掘能力有限，大多数事件依赖关系不确定，导致测试探索空间依然很大。针对这一问题，典型工具 Dcatch [53] 专门用于自动检测云系统中的分布式并发错误。Dcatch 通过跟踪系统运行时的事件顺序和依赖关系，构建事件依赖图，然后在测试过程中识别和重现潜在的并发冲突。Dcatch 使用动态分析技术，通过分析程序执行路径，识别可能导致数据竞争和死锁等并发错误的关键路径。在运行时，Dcatch 插入探针，实时监控和记录事件顺序，并在测试过程中通过模拟和重现这些路径，系统化地检测和定位并发错误。另外，工具 FlyMC [94] 通过识别系统状态对称性和事件独立性，减少需要探索的状态和事件组合数量。FlyMC 在测试过程中并行处理多个独立事件，进一步加快测试过程。此外，FlyMC 通过并行翻转多个事件顺序，系统化地探索复杂事件交错，从而提高检测效率。这种方法通过减少冗余状态和事件组合，有效缩小了输入空间，提高了测试效率。

然而，系统性探索整个输入空间的测试效率较低，为此，典型工具 Morpheus[84]专注于复杂并发执行路径，利用部分顺序采样技术探索稀有但关键的并发冲突。在测试过程中，Morpheus 实时分析系统运行状态，动态调整事件优先级，并基于关键事件进行调度，从而提高发现并发冲突的概率。通过聚焦稀有事件，Morpheus 有效减少了不必要的测试输入，提升整体测试效率。这类工具通过持续优化并发消息输入空间的探索技术，逐步缓解分布式系统测试中的输入空间庞大问题，显著提高了并发错误检测的效率和准确性。

3.1.4 环境错误注入技术

错误注入技术通过模拟各种故障或异常，评估分布式系统在不同异常条件下的表现和鲁棒性。该方法通过引入预定义故障，如网络延迟、数据包丢失、节点故障、资源枯竭等，测试系统在故障场景下的反应、恢复机制和安全性。错误注入通常在系统组件、网络通信或数据流中引入问题，如修改消息内容、模拟节点故障或调整网络参数，从而揭示在特定故障场景下的安全缺陷，如异常处理不当导致的内存问题。由于分布式环境复杂且交互点众多，探索庞大的故障输入空间成为关键挑战。根据探索方式，错误注入技术分为三类：随机故障注入、人工定义故障模型和反馈引导的故障序列生成技术。

随机故障注入技术：考虑到对分布式系统测试来说，可注入的故障空间过于庞大。因此一个简便而高效的策略是在系统的随机位置，于随机时刻注入预先设定好的故障。

Han S 等人早在 1995 年首次提出了集成的分布式系统故障注入环境（DOCTOR）方法[30]，该方法能够在测试环境中随机注入处理器、内存和通信三类故障，并自动收集系统性能数据，评估不同负载下的系统可靠性。故障类型可为永久、短暂或间歇性。随着分布式环境故障种类的增加，支持更多故障类型成为后续技术的发展目标。由 Netflix 开发的故障注入工具 ChaosMonkey[59]最初最初用于测试其在 AWS 云环境中的服务弹性，通过随机终止虚拟机或容器模拟故障，帮助识别和修复潜在弱点。ChaosMonkey 现已成为 Netflix Simian Army 工具套件中的关键部分，专注于模拟 AWS 分布式应用在节点下线时的服务稳定性。相比之下，Alibaba 开发的 ChaosBlade[56]提供了更广泛的故障注入类型，支持物理主机（CPU、磁盘、网络等）、Kubernetes（Pod、节点、容器等）和 Java 应用（JVM、Java 代理等）的故障模拟。ChaosBlade 不仅支持资源层面的故障注入，还能在应用层面进行精细化测试，如模拟方法延迟或变量篡改，提供更丰富的测试场景和细粒度的故障模拟能力。

人工定义故障模型：尽管随机故障测试技术取得了不错的成就，该方法在巨大的故障空间中的探索效率很低。而我们可以利用目标待测系统的一些特征信息来缩小探索空间，提高测试效率。因此一些测试工具提供高度可扩展的测试框架，允许用户使用特定领域语言（DSL）,根据用户自己对待测分布式系统的理解，定义待测系统所要注入的系统故障以及对应的探索策略。

Jepsen[91]，作为业界广泛使用的分布式系统故障注入测试工具，以其灵活的框架设计和全面的 API 接口，赋予了测试人员高度自定义故障注入策略的能力，以满足不同分布式系统测试的复杂需求。通过编写

Clojure 语言的代码脚本及相应配置文件，测试工程师可以详细指定欲模拟的故障类型、注入时机、持续时长及其影响范围，从而准确模拟出网络分区、节点故障、消息丢失等一系列真实世界中可能遇到的故障场景。这种方法不仅使测试过程更加灵活和定制化，还能够针对特定的系统特性和预期的故障场景进行深入测试，有效地揭示潜在的系统缺陷和弱点。Jepsen 的高效测试能力被广泛应用于各类分布式系统，尤其是分布式数据库的深入测试，成功揭示了许多关键的系统代码缺陷。

而考虑到部分系统缺陷是由多种故障组合发生事触发的，针对多个故障的组合空间输入庞大挑战，Joshi P 等人提出了 PreFail[99]，一个可编程的故障注入工具。PreFail 需要测试人员能够编写特定的故障组合策略，以缓解多个故障组合带来的输入空间庞大问题。为了实现可编程性，PreFail 采用了机制和策略分离的经典原则，将故障注入框架解耦为两部分：故障注入引擎和注入驱动程序。故障注入引擎负责在被测系统中注入故障，而注入驱动程序则根据测试人员指定的策略决定故障注入的位置。PreFail 对引擎和驱动程序进行了抽象描述，包括故障注入点、故障执行任务以及测试环境配置等。测试人员可以使用这些抽象在他们的策略中进行快速建模，有效应对多个故障组合的复杂测试需求。

反馈引导故障序列：虽然人工定义的模型可以有效缩小探索空间、提高测试效率，但这种方法通常需要耗费大量人力资源，且构建的模型难以在不同系统间迁移应用。为解决这一问题，一些分布式系统的故障注入工具引入了自动化探索策略。它们将每次测试中触发的一系列故障视为故障序列，并自动生成这些序列作为测试输入。这些工具在测试过程中实时收集并分析系统的反馈信息，根据反馈动态调整故障序列的生成，以更有效地扩展测试状态空间，从而发现更多潜在问题和缺陷。

故障注入工具 FATE[101]在 2011 年首次引入暴力搜索方法来探索故障序列空间。传统暴力搜索通过系统地枚举所有可能的故障组合，逐步增加故障序列的复杂性。为优化效率，FATE 将搜索空间分为前缀和后缀两部分：前缀是固定的操作步骤，后缀则是故障注入点和恢复操作的组合，从而有效减少了组合数量。此外，FATE 引入了“工作流模型”技术，动态识别关键路径和节点，优先测试关键故障序列，最大化测试覆盖率并提升效率。FATE 在 HDFS 和 ZooKeeper 等分布式系统上进行了多重故障测试，探索了超过 40,000 个故障场景，发现了 67 个鲁棒性缺陷。

尽管优化后的暴力搜索在分布式系统缺陷发现上取得了一定成果，但对于需要更长故障路径的深层缺陷，其效果有限，因为生成长路径的故障组合序列消耗大量资源。为提升测试效率，Gao Y 等人开发了 CrashFuzz[76]，一款基于代码覆盖率引导的故障注入工具。CrashFuzz 通过将执行路径上的故障点按顺序排列，并以二进制方式标记其激活状态，形成表示故障序列的二进制字符串。初始阶段所有故障点均未激活，生成序列<000…0>并放入种子池。随后，通过变异种子池中的序列生成新的故障序列，并根据代码覆盖率提升情况选择保留新种子序列。通过这种启发式搜索，CrashFuzz 不仅提升了缺陷挖掘效率和代码覆盖率，还在 HDFS 和 ZooKeeper 等分布式系统中发现了 4 个新的鲁棒性缺陷。

尽管基于代码覆盖率的反馈机制提升了测试覆盖率，但在故障注入测试中，这种方法未必高效，因为其核心是检测分布式系统的故障处理逻辑，而非其他业务逻辑。仅依赖代码覆盖率可能导致大量与故障处理无关的代码被探索，降低了效率。为解决这一问题，Mallory[90]测试工具在系统中插入探针，实时捕捉关键事件信息，如消息传递和节点状态变化，构建“happen-before”图以描述事件间的因果关系。基于该图，Mallory 识别可能导致系统缺陷的事件顺序，并采用 Q-learning 强化学习优化故障输入序列。成功触发新系统行为的序列获得正向奖励，反之则获得负向奖励。通过迭代，Mallory 逐步识别最可能触发缺陷的故障序列，提升了测试生成的质量。Chronos[21]认为在分布式系统中深层路径的故障处理逻辑，通常因较少激活而更有可能隐藏缺陷。Chronos 实时计算执行路径上故障点的深度，优先选择深层路径的故障序列作为测试迭代的种子。在分布式系统的超时鲁棒性缺陷测试中，Chronos 的表现优于 Fate 和 CrashFuzz。

3.2 系统状态感知技术

系统状态感知技术是确保测试有效性的核心，起着承上启下的关键作用。正如图 5 所示，系统感知器实时监控分布式系统的实际行为状态数据，这些数据一方面作为反馈信息，用于优化测试输入生成，应对庞大

的输入搜索空间；另一方面，为缺陷判断准则提供精准依据。分布式系统代码逻辑复杂，涉及业务逻辑、心跳机制、分布式共识机制、数据容灾等系统稳定性功能。在面对百万级代码的分布式系统时，如何高效定位目标行为状态成为测试工具的挑战。目前，主流的分布式系统状态感知技术主要包括三类：（1）基于日志分析的状态感知：通过分析系统生成的日志信息来感知系统状态。（2）基于源码的插桩感知：在源码中插入桩代码，系统运行时收集状态信息。（3）基于 API 的动态获取：通过系统 API 实时查询获取系统关键状态信息。

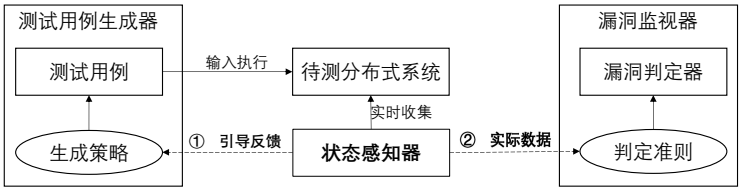


图 5 分布式系统动态测试核心流程

日志分析： 图 6 展示了分布式系统动态测试中基于节点日志信息的状态感知流程。首先，通过多个日志收集器实时收集各节点的日志信息，并进行统一格式化，方便后续分析。接着，各日志收集器将日志汇总传入工具（如 Logstash）进行解析，过滤无关信息并提取关键内容。日志处理可以通过预定义规则自动化，也可借助 AI 模型辅助分析。最后，动态测试工具基于提取的关键状态信息感知系统的实时状态，并据此分析系统行为，判断其是否异常。

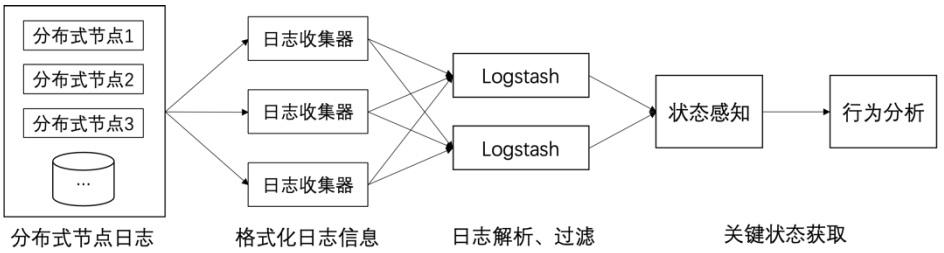


图 6 基于日志分析的状态感知基本流程

基于日志分析的系统感知技术实现相对简单，因其非侵入式设计不需要访问待测系统的源码，被广泛应用于各种动态测试工具中。以典型工具 CrashTuner [20] 为例，它通过解析运行时产生的日志数据，识别和收集系统中关键节点的引用变量及其对应的运行时值。具体来说，CrashTuner 的开发者首先人工详细审查了被测系统生成的所有日志条目，发现大多数分布式系统采用 Log4j、SLF4J 等流行的日志记录库，这些库定义了 fatal、error、warn、info、debug 和 trace 等级别的通用日志接口。因此，CrashTuner 通过简单匹配的方法查找调用名称与这些日志接口名称，高效地识别出系统的关键日志记录点。CrashTuner 为每个分布式节点分别部署了一个日志分析器实例，这些实例能够并行地监视并收集关键变量信息，实时从日志数据中挖掘出分布式系统的元信息(即引用系统高级状态信息的变量：例如，节点或任务的实例)及其之间的依赖关系，从而有效推导出系统的高层状态信息。基于这些实时信息，CrashTuner 一方面决策引导可能触发错误的故障注入点，另一方面识别日志中的异常信息来判断是否出现缺陷。

然而，基于日志分析的状态感知技术存在信息获取不全的问题，往往只能捕捉较为宏观的系统状态，无法覆盖所有细节状态，特别是分布式协议中的细微状态转移等内部状态信息。为了弥补这一不足，可以采用代码级监控技术，通过在代码中插入监控点（插桩）来实时追踪系统的执行情况。这种方法能够记录代码的执行路径、分支选择以及函数调用等详细信息，提供更丰富的系统运行细节。借助这些数据，测试工具能够更加全面地了解系统的行为状态，从而提高测试的精确性和有效性。

代码插桩：图 7 展示了分布式系统动态测试工具中代码插桩的基本流程。首先，获取待测系统源代码并进行静态分析，识别基本代码块和控制流分支。随后，利用编译器技术进行插桩，在源代码中嵌入用于跟踪

执行情况和关键状态的特殊代码段，生成增强版可执行文件。运行时，这些代码段在特定操作执行时激活，记录测试输入的执行细节，包括执行路径和关键状态变化。这些详细的执行信息帮助测试工具实时分析系统行为，识别异常状态或行为。通过分析代码覆盖率数据，测试工具评估测试输入的覆盖范围，并优化测试策略，生成新的测试输入以更全面测试系统，挖掘潜在缺陷。根据插桩流程不同，常见插桩工具分为三类：源码静态插桩、编译器插桩和二进制插桩。

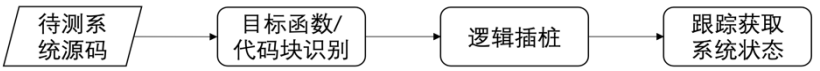


图 7 动态测试工具进行代码插装的基本流程

早期的插桩工具直接在源代码中添加或修改代码，以在程序执行时收集实时信息，如执行路径和变量状态。针对 C/C++程序代码，典型的工具有 Gcov/Lcov[100]，它是 GNU 工具链的一部分，通过在源代码中插桩来收集程序执行信息，帮助实时获取分析测试覆盖率指标。另一个工具是 JaCoCo[68]，它是为 Java 程序设计的代码覆盖率库，通过在字节码层面插桩，支持 Eclipse 和 Maven 等集成，常用于 Java 项目的测试覆盖率分析。这种方法提供了高度的灵活性，但可能会影响原始代码的结构和性能。

不同于源代码修改，常用编译工具 GNU Compiler Collection (GCC) 提供的 `-finstrument-functions' 选项是一种编译时插桩技术，允许开发者在每个函数的出入口自动插入用户定义的监控逻辑。在编译阶段编译器会在每个函数的开始处调用“__cyg_profile_func_enter”接口，在函数返回前调用“_cyg_profile_func_exit”接口。这两个接口需要由开发者实现具体的监控逻辑，它们的参数包括当前函数和调用者的地址，使得动态测试器可以跟踪函数的调用情况，如调用次数、执行路径等。这项技术对于测试分析、调试和理解程序行为非常有用，尤其是在需要深入分析程序执行细节的场景中。通过这种方式，动态测试工具可以在不修改源代码的情况下，获得关于程序运行时行为的详细信息，从而帮助动态优化测试流程和诊断系统中代码缺陷。

不同于源码静态插装和编译器插桩，Intel PIN 是一个强大的动态二进制插桩框架，允许测试器在不访问源代码的情况下对运行中的程序进行监控和分析。作为一个运行时重写工具，PIN 提供了一种灵活的方式，将自定义的分析代码注入到目标应用程序的二进制文件中。开发者可以使用 PIN 跟踪程序的执行，收集各种运行时信息，如函数调用、内存访问模式、分支预测和指令使用情况等。PIN 的插桩在程序运行时动态完成，这意味着可以在不重新编译目标程序的情况下对其行为进行深入分析和调整。这使得 PIN 成为测试分析、程序调试、软件安全检测等领域的有力工具，特别适用于复杂软件系统和闭源应用程序的分析。

虽然代码插桩技术在获取系统运行信息方面表现出色，但其侵入性质要求测试人员必须能够访问甚至修改待测分布式系统的源代码或二进制程序，这对测试对象的控制权要求较高，从而限制了该技术的应用范围和普适性。考虑到在许多实际测试场景中无法获得目标源代码或二进制程序，为了降低测试工具对被测试系统的依赖，减少测试的侵入性并提升工具的适用性，一种可行的策略是利用分布式系统提供的 API 接口来动态捕获关键的系统状态信息。

动态接口感知：图 8 展示了分布式系统动态测试工具中动态接口状态感知的流程。首先，收集待测系统的 API 清单，并根据测试目标筛选关键 API。在测试执行过程中，状态收集器动态调用这些 API，实时捕获系统的关键状态信息。随后，这些状态信息用于行为分析，识别系统行为异常并进行缺陷判断。通过此方法，测试工具能够高效监控系统状态，及时发现和定位潜在缺陷，提升测试效率和准确性。

典型动态测试工具 Tyr[79]是一个检测区块链系统中共识缺陷的工具，它通过利用系统内置的区块查询 API 接口来动态监控区块链的共识状态信息。在测试开始时，Tyr 会与区块链网络中的各个分布式节点建立连接，利用节点的 socket 服务接口，实时收集包括区块高度、哈希值、节点成为领导者的次数以及区块内存储的交易数据等关键行为状态信息。随后，Tyr 利用这些数据计算不同节点间的共识差异度，并使用这一差异作为测试流程的引导反馈，生成拜占庭消息。此外，漏洞检测器基于这些实际状态信息，检测待测系统的共识差异是否在较长时间内保持较高水平。如果系统长时间(超过 6 个区块共识时间)处于异常不一致状态，则 Tyr 判断系统可能未能达成正常共识状态，指出系统中可能存在的一致性缺陷。

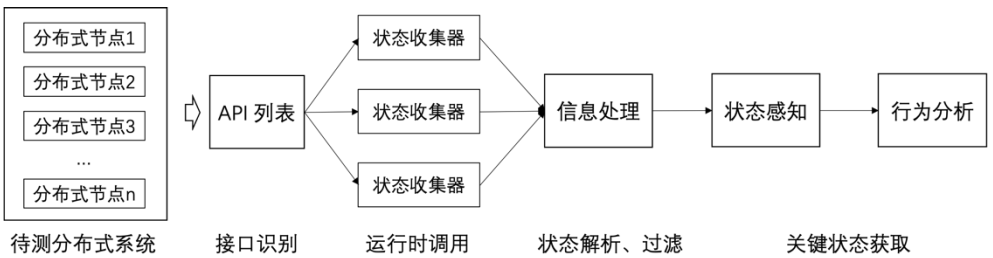


图 8 测试工具动态接口状态感知的基本流程

3.3 判断准则构建技术

测试准则构建是动态测试中用于识别问题的基础，尤其在自动化测试中至关重要。它提供标准来评估测试输入执行结果是否符合预期，直接影响能发现的问题类型及诊断准确性。作为分布式系统动态测试的核心，测试准则构建技术应对各种特性缺陷，优化缺陷识别过程，提升挖掘率和测试效率。该技术通过深入分析待测系统的特性和潜在缺陷，建立一套判定规则，判断系统执行结果或状态是否异常，从而提高测试的有效性。

测试准则构建技术的设计包含两个关键目标：（1）深入分析并分类目标缺陷，通过审查系统源代码、历史缺陷记录和相关文献，理解缺陷的特性和成因，确定测试应关注的缺陷类型和场景，为准则制定奠定基础。（2）基于缺陷的理解，制定具体测试准则或规则，用于测试过程中评估系统是否出现异常。这些准则包括对返回值的判断、关键状态的检查及异常处理评估，需充分考虑分布式系统的特性和缺陷特点，确保能有效捕捉潜在问题。目前，分布式系统动态测试常用准则包括：基于反馈响应的崩溃检测、差分测试和蜕变测试进行缺陷判断、用户建模或系统特征总结进行异常识别，以及基于大数据分析的离群点检测等策略。这些方法能够有效提升测试的覆盖率和缺陷挖掘能力。

3.3.1 崩溃识别技术

崩溃缺陷指的是由于代码缺陷、资源管理不当等原因，导致某个关键组件、进程、节点甚至整个系统突然停止工作的安全缺陷。这类缺陷可能导致数据丢失、服务中断，甚至整个系统的崩溃，严重影响系统的可用性、可靠性和数据完整性。在分布式环境中，崩溃缺陷尤其严重，因为一个组件的失败可能通过依赖链影响其他组件，放大故障的影响范围。此外，分布式系统通常承载着关键业务，崩溃缺陷的发生不仅会导致直接的经济损失，还可能损害企业的声誉。

崩溃缺陷的识别相对直接，因为它的表象十分明显，即判断原有的进程、节点、服务是否还持续在线。因此，许多分布式系统动态测试工具（如 ECFuzz[70]、LOKI[66]、Chronos[21]）在系统测试时都会进行崩溃缺陷的检测。针对进程崩溃问题，动态测试工具通常在系统启动时记录下关键进程号，并在整个测试过程中实时监控操作系统（OS）抛出的系统崩溃信号是否在这些关键进程列表中。针对节点崩溃问题，工具会在系统启动时与各个分布式节点建立长链接，并周期性地发送心跳包，等待回包，若长时间没有心跳回包，则判断对方节点处于崩溃状态。对于服务崩溃问题，工具通过模拟发送用户级请求并监控是否收到正常响应来进行测试，若服务无法返回预期的响应，则可能指示服务已处于崩溃状态中。

一些内存安全问题，如缓冲区溢出、栈溢出、使用已释放的内存和内存泄漏等，可能不会立即导致程序崩溃，使得它们难以被直接观测识别出来。因此，在针对 C/C++ 语言实现的系统进行测试时，通常会借助 AddressSanitizer（ASan）[95]工具来辅助识别这类内存安全问题。ASan 在编译阶段向程序源代码中插入额外的检查代码，用于监控内存访问操作。具体来说，ASan 通过维护影子内存（shadow memory），记录每个内存字节的状态，追踪程序对内存的使用情况。当程序运行时，这些插入的检查代码会检测潜在的内存错误，如堆栈溢出、使用后释放、双重释放等。如果检测到非法访问或未初始化的内存使用，ASan 会立即报告错误，提供详细的错误信息和调用栈，帮助开发者快速定位和修复问题。这种方法在保持较低运行时开销的同时，能有效检测出各种复杂的内存错误。

3.3.2 差分检测技术

基于差分测试构建的测试准则：相较于崩溃问题，系统逻辑缺陷更难检测，因为它们通常不会导致系统的显式崩溃或异常。在分布式系统中，差分测试是检测逻辑缺陷的常用方法之一。差分测试的核心思想是使用相同的测试输入传递给不同的分布式系统实现，然后对比其输出结果。如果结果出现不一致，通常表明存在潜在缺陷。差分测试的优势在于无需了解系统的内部实现细节，通过对比多个系统实现的输出行为即可发现问题。图 9 展示了差分测试的基本流程，该方法通过对比不同实现的输出，有效检测出代码逻辑缺陷。

典型测试工具 DiffStream[104]采用差分测试来检测分布式数据流处理系统的逻辑问题。其主要关注点是检查两个实现对给定输入流是否产生等价的输出流，即允许逻辑上独立的数据项顺序不同。由于分布式系统的数据处理通常是并行的，在差分测试过程中，两个实现可能会以不同的速率、异步且乱序地产生事件。为了解决事件输出的乱序性并进行精准的等价输出判断，DiffStream 提出了一种最优在线匹配算法，用于高效、快速地逐步比较两个输出流的等价性。

针对区块链系统的动态测试工具 Fluffy[105]使用差分对比的测试准则来判断不同以太坊客户端的输出结果。具体而言，Fluffy 首先基于以太坊客户端虚拟机的执行状态，通过动态变异生成大量测试交易序列。然后，将这些序列作为测试输入，分别输入到不同版本的以太坊实现中，包括 Go 语言版本的 go-ethereum 和 Rust 语言版本的 OpenEthereum。通过比较这两种客户端对同一交易序列的执行结果，Fluffy 能够识别出结果不一致的情况，从而指出至少一个客户端实现中可能存在的代码缺陷。

考虑到许多分布式系统只有一种代码实现，无法直接构造差分对比对象，典型工具 Mocket[51]设计了基于系统设计与实现的差分对比准则。在设计阶段，分布式系统通常使用形式化语言（如 TLA+[52]）对关键逻辑行为进行建模，并在形式化验证通过后开始代码开发。然而，分布式系统的逻辑复杂性常常导致代码开发过程中引入缺陷，造成实现逻辑与设计逻辑的不一致。为了解决这一问题，Mocket 建立了设计模型与代码实现中关键行为状态的映射关系，并使用状态转移边覆盖引导的模糊测试技术，生成大量测试用例，尽可能触发并识别代码实现与模型设计的不一致性，从而检测出代码逻辑缺陷。然而，由于设计模型中的逻辑通常只是代码实现功能的一个子集，许多非模型中的逻辑缺陷可能会被遗漏。

尽管差分测试是一种有效的测试方法，但它也有一些缺点和限制。首先，差分测试只能测试具有相同功能目标的分布式系统逻辑。由于各个分布式系统通常具有自己设计上的侧重点，例如同为分布式文件系统，CephFS 通过提供数据复制和条带化来保障数据的高可用性和耐用性，适合需要高吞吐量和低延迟的应用；GlusterFS 侧重于灵活性和简易性，通过其强大的卷管理功能，能够轻松扩展存储容量和性能，适合云存储、媒体流和内容分发网络；而 HDFS 侧重于支持大规模数据处理工作负载，适合大数据分析、机器学习和数据湖场景。尽管这些系统的主要功能目标都是分布式文件存储，但由于侧重点截然不同，差分测试仅能覆盖它们功能相同的部分，而无法检测到各自特有的功能逻辑，从而遗漏大量的潜在缺陷。此外，如果用于对比的多个系统实现都存在相同的代码缺陷，差分测试同样会造成漏报问题。

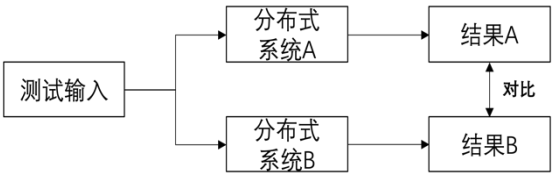


图 9 分布式系统差分测试示意图

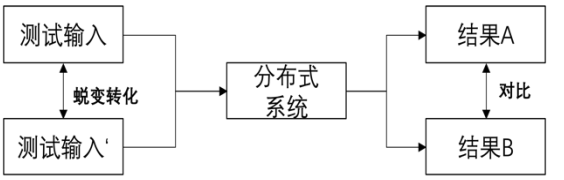


图 10 分布式系统蜕变测试示意图

3.3.3 蜕变测试准则构建

基于蜕变测试构建的测试准则： 蜕变测试（Metamorphic Testing）通常应用在难以找到用来对比的差分对象的测试场景上。蜕变测试基于软件应该遵循的某些性质或规律（称为蜕变关系），通过对原始输入和经过某种方式变换后的输入进行处理，然后比较输出结果是否符合这些蜕变关系，从而判断程序是否存在缺陷。

蜕变测试的核心难点在于需要根据分布式系统的功能和需求，确定一组蜕变关系。这些关系是基于系统应遵循的行为特性定义的，例如，系统的输出应该对某些输入变化保持不变，或者某些操作的序列不应影响最终结果。以数学公式上的蜕变关系 sin(x) = -sin(-x)为例。我们的测试输入 x 和 -x 需满足结果 A = -B。然而，这样的关系的寻找和确定往往需要领域专家深入的知识。如图 10 所示，为分布式系统建立蜕变关系的一种常见方式是通过创建具有等效语义的不同分布式系统输入，并比较它们的响应行为、输出结果。

动态测试工具 TEA-Cloud[83] 使用蜕变测试准则来自动判断分布式系统的配置处理逻辑行为是否合法。TEA-Cloud 针对分布式系统的性能与资源配置、资源使用之间的关系，设计了三个通用蜕变转化关系。蜕变关系一：给定两个分布式系统 DFS 和 DFS' 以及两个工作负载 ω 和 ω'，如果 ω 和 ω' 相等，并且 DFS 使用的 CPU 性能大于 DFS' 使用的 CPU 性能（其他方面相同），则执行系统 DFS 执行 ω 所需的时间必须小于或等于执行 DFS' 执行 ω' 所需的时间，或者在相同时间内 DFS 上成功处理的 ω 的用户请求数量必须大于在 DFS' 上成功处理的 ω' 的用户请求数量。蜕变关系二：给定两个分布式系统 DFS 和 DFS' 以及两个工作负载 ω 和 ω'，如果 ω' 是对 ω 使用标准排序后构建的 ω 的序列，表示为 sort(ω)，则产生按时间顺序排序的序列 $\omega' = \{((t_1, ts_1), (t_2, ts_2), ..., (t_n, ts_n))\}$，则对于所有 $1 <= i < n$，我们有 $ts_i <= ts_i + 1$，且 DFS 和 DFS' 相等，则在系统 DFS 上成功处理 ω' 的用户请求数量必须等于 ω 中的用户请求数量。蜕变关系三：给定两个分布式系统 DFS 和 DFS' 以及两个工作负载 ω 和 ω'，如果 DFS 和 DFS' 相等并且负载 ω 包含 ω'，则系统 DFS 处理负载 ω 所需的总成本必须小于或等于系统 DFS' 处理 ω' 所需的总成本。基于这三个蜕变关系，TEA-Cloud 构造了大量的测试输入，进行蜕变结果对比，成功挖掘了许多潜藏在分布式系统中的逻辑缺陷。

然而，在 TEA-Cloud 测试过程中，蜕变结果不符合预期不一定是系统缺陷所导致的，可能是测试环境中的不可控因素带来的扰动，从而导致误报。为了解决这一问题，另一典型动态测试工具 ChaT[77]也基于系统配置和系统性能之间的关系，构造了四种通用的蜕变关系。ChaT 使用几种机器学习技术（隔离森林、一类支持向量机、局部离群因子和鲁棒协方差算法）对这些蜕变关系进行验证。通过对大量测试过程中产生的运行时数据进行分析，ChaT 能够辅助判断蜕变结果的差异是由不可控实验环境干扰带来的还是由系统自身缺陷导致的，从而有效减少测试工具的误报。

3.3.4 用户自定义检测模型

用户自定义判定模型：考虑到蜕变测试通常难以精准构建蜕变关系，目前一些分布式系统的动态测试工具往往采取更灵活的策略来应对缺陷判断准则的建立问题。这些工具提供了一个通用的测试框架，并通过设计用户友好的接口，使得用户可以利用特定领域语言（DSL）定制化测试准则。这种方法允许用户根据对待测系统的深入理解，自定义针对系统特定关注点的缺陷检测规则，从而在动态测试过程中有效识别潜在缺陷。

作为使用最为广泛的分布式系统测试工具之一，Jepsen[91] 凭借其灵活的框架设计和全面的 API 接口，支持测试人员自定义系统的缺陷判断模型。通过编写 Clojure 语言的代码脚本及相应配置文件，测试工程师可以详细定义待测系统的异常行为判断条件。在测试过程中，Jepsen 会实时检测这些判断准则，并在识别出行为异常时及时输出缺陷报告。下图 11 展示了一个 Jepsen 的缺陷判断模型样例，`ExampleChecker'定义了一个简单的检测器，对比测试执行前后读取的返回结果，用于检查分布式系统的读取操作是否一致。

Deligiannis P 等人提出的动态测试工具[107]通过使用 P#[102]编程语言（一种对 C#语言的扩展）来检测分布式系统的关键属性。P#语言支持用户对属性进行正确性判断建模，模型需要指定两种属性规范：（1）安全属性规范概括了源代码断言的概念，安全违规是导致错误状态的有限轨迹。P#支持使用断言来指定 P#机器本地的安全属性，还提供了一种使用安全监视器来指定全局断言的方法。（2）活性属性规范表示非终止性，一个典型的活性缺陷是死循环执行路径。通常，活性属性通过时间逻辑公式来指定。基于这两种属性规范，该工具在测试过程中实时监控模型中指定的属性，任何违反这两种属性的系统执行情况将被识别为系统逻辑缺陷并记录下来。另一个类似的动态测试框架 Frisbee[81]，提供了一种声明性语言和相关的运行时组件，用于在 Kubernetes 上测试分布式应用程序。首先，Frisbee 需要用户提供一个自定义模型，该模型描述被测系统

的关键状态机和依赖环境及其启动流程。接着，Frisbee 会自动与 Kubernetes 容器对接，在容器中部署必要的软件，启动待测系统，并根据状态机模型执行测试。通过持续监控系统运行状态，Frisbee 自动检查与预期行为的偏差，从而有效地发现和报告系统中的潜在缺陷。

```
(ns jepsen-example
...
(defrecord ExampleChecker []
  checker/Checker
  (check [this test history]
    (let [reads (->> history
                     (filter #(= :read (:f %)))
                     (map :value))]
      (if (apply = reads)
        {:valid? true}
        {:valid? false, :error "Values are not consistent!"}))))
...
)
```

图 11　Jepsen 缺陷检测模型示例

3.3.5　基于系统特征总结的测试准则

基于系统特征总结的测试准则：用户自定义判断模型面临高人力成本、长学习与建模周期，以及缺乏通用性等挑战，限制了其广泛应用。鉴于相同类型的分布式系统通常共享一些基本特征和功能目标，可以通过归纳总结这些共性特征来构建通用测试准则。该方法包含两个核心步骤：首先，分通过分析待测系统的历史缺陷记录，人工总结出导致缺陷的共性特征和根本原因；其次，从系统的设计和架构出发，研究设计文档、架构图及相关技术规范，提炼系统关键属性（如一致性、原子性、隔离性等），从中制定通用测试准则。

针对分布式系统升级过程中的缺陷，Zhang Y 等人分析了 8 个广泛使用的分布式系统中 123 个实际升级失败的历史报告，详细总结了升级失败的严重性、根本原因、暴露条件及修复策略。基于此，他们设计并实现了缺陷判断器 DUPTester[65]，用于检测跨版本的系统不兼容问题。DUPTester 主要关注两类数据不兼容问题：一是序列化库定义的数据，当新版本修改消息或文件格式时，可能导致升级失败；二是枚举类型数据，当升级时枚举类中增加新变量，导致后续变量索引变化，进而与原版本语义不兼容。通过检测这些数据的兼容性，DUPTester 在主流分布式系统中累计发现了 800 多个由于升级导致的不兼容逻辑缺陷，有效提高了系统升级的稳定性。

针对分布式网络故障的特征分析，Alquraan A 等人提出了动态测试工具 NEAT[106]。该工具首先进行了全面的分布式缺陷调研，分析了广泛使用的分布式系统中的 136 个严重系统故障，发现其中 25 个故障是由网络分区故障引起的，并且这些故障大多会导致灾难性的影响，如数据丢失、已删除数据重新出现、锁失效和系统崩溃等。基于这些历史缺陷报告，NEAT 确定了故障触发的执行顺序、时序和网络故障特征，包括特定的事件执行顺序、网络隔离的持续时间、网络延迟和数据包丢失率等。随后，NEAT 依据这些特征构建了自动化的网络分区缺陷判定器，大大简化了测试流程。NEAT 在 7 个主流分布式系统上进行了评估验证，累计发现并报告了 32 个缺陷。

针对去中心化区块链系统共识过程中的缺陷，Chen Y 等人[79]通过分析主流区块链系统（例如以太坊、Fabric、EOS 等）的共识设计文档，总结了四个通用的共识缺陷判断准则：（1）活性缺陷准则：所有合法交易最终都应被执行并存入区块链中；（2）安全缺陷准则：所有不合法交易（如双花交易）最终不应被执行或存入区块链中；（3）公平性准则：所有参与共识的节点当选 leader 节点的概率应公平公正；（4）完整性缺陷准则：整个区块链网络应处于连通状态，不存在硬分叉。基于这四个缺陷判断准则，Chen 等人设计实现了区块链共识逻辑缺陷检测器 Tyr。对于第一和第二个缺陷准则，Tyr 基于区块链的全局状态信息，构建一系列合法和不合法交易，并创建预期状态列表。在测试过程中，Tyr 实时监控全局状态信息，判断状态是否符合预期值。针对第三和第四个缺陷准则，Tyr 通过实时获取的区块链关键状态信息，判断分布式节点的各自当选

leader 数与区块高度差异是否正常。借助这四个缺陷判断准则，Tyr 在 6 个主流区块链系统上成功发现了 20 多个新的共识逻辑缺陷。

3.3.6 基于大数据分析的性能测试准则

在分布式系统中，尽管一些性能缺陷不会直接导致系统停止工作或无法提供服务，但其对系统处理性能的严重影响不容忽视。与可直接判定的崩溃或逻辑错误不同，性能问题通常在系统负荷增加或特定环境条件触发时逐渐显现，这增加了早期发现性能问题的难度。此外，鉴于分布式环境下性能波动在一定程度上是正常且可接受的，如何准确界定“性能下降”的标准成为关键挑战，主要涉及两个问题：（1）性能降低到何种程度才算是一个问题；（2）性能降低状态持续多长时间才算是一个真缺陷。最简单的方法是设定阈值来判断性能下降是否达到了关注的级别，然而阈值判断会存在许多误报：某些性能下降可能是由于网络波动等分布式环境因素造成的暂时性波动。因此，目前测试工具主要使用基于大数据分析的模型来识别性能下降的模式与特征。这些模型通过分析系统运行时的海量数据，提取出性能波动的规律，并结合机器学习算法进行预测和判断，从而更准确地识别真正的性能缺陷，减少误报，提高测试效率。

典型检测工具 IASO [97]建立了一个响应超时模型来识别分布式系统的性能下降问题。具体来说，IASO 构建了一个超时评分算法，通过持续记录各节点间的回复超时次数和正常回复次数，计算每个节点的连接请求响应率，并基于此计算性能评估分数。然后，将每个节点的性能评分置于一个响应超时集合中，使用 DBSCAN [55]数据分析聚类算法自动划分这些性能评分，识别并排除噪声点，并将噪声点视为性能缺陷点。然而，该方法未能排除真实网络负载和用户请求负载变化对分布式节点响应的影响，导致许多误报。

为了解决这一问题，Lu R 等人提出了一个基于大数据分析的性能缺陷检测器 Perseus [78]。Perseus 首先收集每个节点的实时运行状态数据，包括每个磁盘的吞吐量、写延迟以及对应的时间戳。然后，使用主成分分析和基于密度的噪点空间聚类法对离群点进行数据清洗，去除网络波动、负载变化等外界因素对模型的影响。接着，基于清洗后的数据，通过多项式回归获得一个检测模型，并使用预测上界作为性能下降缺陷的检测阈值，在原数据上进行检测，超过这个上界的被认为是进入离群状态。最后，通过一个滑动时间窗口计算每个磁盘处于离群状态的时间，并根据时间划分等级，计算各自的风险值。如果一次离群状态持续时间很长，且每天进入离群状态的次数很多，则被认为是高风险，是一个潜在的性能缺陷。

4 典型分布式系统动态测试工具测评

目前存在许多分布式系统动态测试工具，它们针对各自关注的系统缺陷，设计了不同的测试策略，并在各自的评估实验中表现出优异的性能。然而，许多动态测试工具仅关注分布式系统的一部分代码逻辑，只针对目标模块代码进行测试评估，缺乏对分布式系统整体测试效果的感知。此外，由于不同工具评测方式的多样性和不统一，评估的有效性受到威胁，使得工具的使用者和研究者对工具的效果缺乏准确的认知。因此，本章将从对目标待测系统的整体覆盖率和发现的缺陷数量这两个重要维度，对近年来典型的分布式系统动态测试工具进行统一直观的测评，进而总结现有工具的不足，提出未来可能的改进方向。

4.1 实验设置

对比工具。我们首先按以下三个标准选取了一些分布式系统动态测试工具：（1）最新的（近三年发布），（2）工具代码开源，（3）适配性和易用性较强。随后，根据测试工具主要关注的输入维度，将其分为四类，在每个维度上取最新发布的两个测试工具：针对配置输入生成的测试工具 CTests[22]和 ECFuzz[70]；针对用户请求负载合成的工具 DUPTester[65]和 Mocket[51]；针对节点间协作消息生成的工具 LOKI[66]和 Tyr[79]；以及针对环境故障注入的测试工具 CrashFuzz[67]和 Chronos[21]。

待测试分布式系统。为了评估各个工具的通用性和测试效率，我们选择了三个流行的开源分布式系统进行评测：两个中心化分布式系统 HDFS（版本 3.0.0）和 ZooKeeper（版本 3.5.6），以及一个去中心化分布式系统 IPFS（版本 1.0.7）。选择这些分布式系统是因为它们在工业界被广泛使用并经过了大量测试，同时也被

大部分研究工作选为测评对象。其中，考虑到测试工具 ECFuzz、CTests、CrashFuzz 和 DUPTester 在论文撰写时仅对 HDFS 和 ZooKeeper 的支持较为成熟，因此我们在 IPFS 系统上根据工具的开源代码进行了手动适配。考虑到 LOKI 和 Tyr 是针对去中心化分布式系统中的拜占庭共识协议进行测试，而 HDFS 和 ZooKeeper 不支持拜占庭容错，我们在将 LOKI 和 Tyr 适配到 HDFS 和 ZooKeeper 上时，去除了工具原本的拜占庭消息包生成，仅生成符合待测系统消息规范的合法通信包。

实验环境。我们在一台运行 64 位 Ubuntu 20.04 操作系统的机器上进行实验，该机器配备了 128 个内核（AMD EPYC 7742 处理器，主频 2.25 GHz）和 512 GiB 主内存。所有测试的分布式系统都运行在 docker 容器中，这些容器可以直接从其官方网站下载。为了进行定量比较，我们为 HDFS、ZooKeeper 和 IPFS 的 docker 容器配置了 6 核 CPU、16GiB 内存和 480GB 固态硬盘。每个待测系统均启动了一个由 20 个节点组成的私有分布式网络进行测试，节点之间的带宽为 10Gbps。所有实验的测试时间均设定为 24 小时，这是因为目前大多数动态测试工具的评估工作都采用 24 小时作为通用的评估时间。

测试过程。图 12 描述了实验的测试过程，具体包括四个步骤：测试环境准备、分布式系统构建、测试执行和结果分析。第一步是环境准备，需要准备测试工具及其依赖环境，下载待测工具源码，配置环境依赖，并通过构建指令生成工具的可执行二进制文件。第二步是分布式系统构建，用于构建分布式系统测试网络。首先获取待测系统的最新版本源码，并根据构建脚本自动生成测试版本二进制及对应的覆盖率收集版本。第三步是测试执行，进行实际测试。先通过构建动态测试工具与待测系统的网络进行预测试。如果预测试存在问题，需返回前两步修改；若预测试通过，则执行 24 小时的持续测试。第四步是结果分析，用以统计分析各个测试工具在不同系统上的测试结果。首先收集待测系统的代码覆盖率数据，这些数据帮助评估各类测试工具的有效性及代码探索程度。除覆盖率数据外，还重点统计各工具挖掘出的缺陷数量。利用工具自带的去重策略进行初步去重后，进行人工分析，进一步定位缺陷根本原因并精细去重。

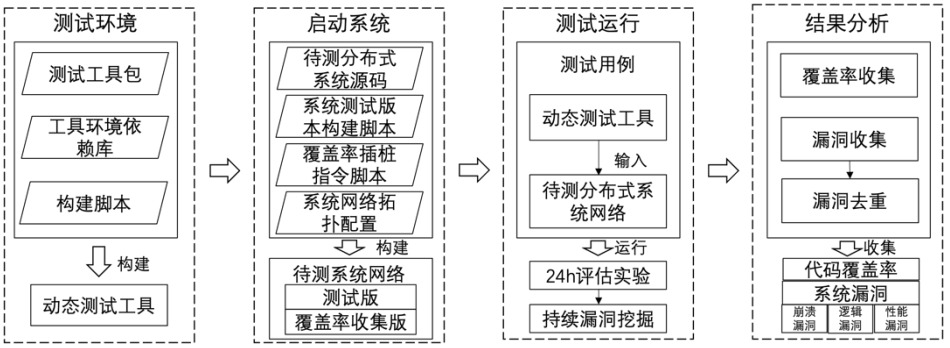


图 12 实验的测试过程

4.2 评估结果分析

4.2.1 工具测试表现

考虑到不同语言实现系统的代码覆盖率收集方法不一样：对于 HDFS、ZooKeeper 等 Java 项目，我们使用 JaCoCo[68]工具收集覆盖率信息。基于 JaCoCo 的代理模式，通过 Java 的 ClassLoader 探针对 HDFS 和 ZooKeeper 进行实时覆盖率信息收集。该方法的覆盖率收集流程主要包含三个主要步骤：1. 启动待测分布式系统时加上“-javaagent: path_to_jacococoagent”选项启动 JaCoCo 的代理模式；2. 在插装后的待测系统上执行测试；3. 测试结束后，使用“java -jar jacococli.jar report path_to_report”生成详细的覆盖率报告。同样，对于 Go 语言项目 IPFS，我们实现了 go-cov[69]的代理模式：在待测系统运行时启动覆盖率收集监听器 cov-server，实时统计覆盖率信息，形成覆盖测试报告。使用“go-cov -report path_to_report”命令查看详细的覆盖率信息，报告内容包括文件覆盖率、函数覆盖率、代码行覆盖率以及分支覆盖率。我们选取了大多数测试工具中最常用的代码行覆盖率和分支覆盖数信息作为测试评估指标，具体的覆盖信息如表 4 所示。

表 4 各动态测试工具在待测分布式系统上的覆盖数据

分布式系统	HDFS		ZooKeeper		IPFS	
工具名	行(%)	分支	行(%)	分支	行(%)	分支
CTests	23.45%	31683	16.15%	12640	14.49%	2011
ECFuzz	24.38%	32588	16.83%	13247	15.02%	2079
DUPTester	27.42%	38194	18.65%	14369	16.58%	2391
Mocket	25.91%	335392	17.48%	13559	15.85%	2396
LOKI	21.09%	29539	13.76%	11381	16.93%	2462
Tyr	21.83%	30523	14.02%	11639	17.59%	2517
CrashFuzz	23.01%	31592	14.23%	11662	15.76%	2399
Chronos	24.27%	32908	15.04%	12298	15.98%	2448

表 5 各动态测试工具在待测分布式系统上的缺陷检测数

分布式系统	HDFS		ZooKeeper		IPFS	
工具名	崩溃缺陷	逻辑缺陷	崩溃缺陷	逻辑缺陷	崩溃缺陷	逻辑缺陷
CTests	2	1	0	1	0	0
ECFuzz	2	1	1	1	0	1
DUPTester	1	2	1	1	0	1
Mocket	1	1	0	1	0	0
LOKI	0	0	1	0	1	1
Tyr	0	1	1	0	1	2
CrashFuzz	1	1	0	2	0	0
Chronos	1	1	1	2	0	1
总缺陷数	4	6	3	4	1	3

此外，考虑到各个测试工具除了针对分布式系统的不同目标逻辑设计了各自的缺陷检测器外，都还支持分布式系统的崩溃缺陷检测。因此，我们收集了各个动态测试工具在 24 小时内发现的缺陷数目，包括崩溃缺陷数目以及逻辑缺陷数目，如表 5 所示。对于崩溃缺陷，我们使用统一的基于错误栈信息的去重方法，进行人工分析以过滤重复的缺陷；对于逻辑缺陷，我们使用各工具自带的去重策略对报告的缺陷进行分析去重。考虑到不同工具找到的缺陷可能会有交叉重复，我们通过对上述工具报出的所有缺陷进行人工分析、分类和去重处理，最终统计出各待测分布式系统基于这些测试工具发现的总缺陷数。

由表 4 中数据可以看出，所有工具在各分布式系统上的代码行覆盖率均不超过 30%，处于较低水平。这主要是因为分布式系统功能复杂多样，但大部分测试工具目前都只关注于其中的某些功能组件进行测试，导致大量丰富的不同组件交互逻辑难以深入覆盖。从表中数据可以看出，用户请求输入生成工具 Mocket 和 DUPTester 总是能覆盖最多的代码行和分支覆盖数。这是因为它们生成的测试输入是与待测系统的功能逻辑相关的，而一个分布式系统中功能处理代码往往占比较多且更容易触发。与 Mocket 相比，DUPTester 在三个待测系统上均表现出更佳的测试覆盖率，这是因为 Mocket 的测试输入是基于分布式系统设计层面建模的，从而会遗漏一些代码实现上的功能逻辑。

此外，系统配置生成工具 CTests 和 ECFuzz 在测试覆盖率上表现出色，这表明分布式系统需要大量代码来处理多变的配置项。与 CTests 相比，ECFuzz 在三个待测系统上均表现出更佳的覆盖率，这是因为它不仅对配置项的语法语义进行了建模，还对不同配置项间的依赖关系进行了深入优化，从而提升了测试效果。错误注入工具 Chronos 和 CrashFuzz 也取得了较高的覆盖率，因为分布式系统中包含大量的错误处理代码，以应对复杂的运行环境。Chronos 相较于 CrashFuzz 覆盖率更高，得益于其基于代码深度覆盖优先的算法，能够更加高效地生成测试输入，探索系统深层路径，提高整体代码覆盖率。最后，消息生成工具 LOKI 和 Tyr 在去中心化系统 IPFS 上取得了较好的代码覆盖效果，但在中心化系统 HDFS 和 ZooKeeper 上表现较弱。这是因为中心化系统的消息传递机制较简单，相关的逻辑处理代码也较少，限制了测试覆盖率的提升。

4.2.2 现有工具不足

从表 4 和表 5 的数据可以看出，现有的动态测试工具的测试覆盖率仍然偏低，且找到的缺陷数量远少于当前分布式系统中的已知缺陷数量。以主流分布式系统 HDFS 为例，根据 HDFS 系统官方修复的缺陷报告，

我们分析并总结了版本 3.0.0 的历史缺陷，其中与配置、用户请求、消息处理、故障处理相关的系统缺陷共有 24 个，而这 8 个典型动态测试工具仅发现了其中的 10 个缺陷，约有 58%的 HDFS 缺陷被这些工具所遗漏。为了进一步分析现有工具的不足，我们对剩余的 14 个 HDFS 缺陷进行了初步人工分析，发现这些缺陷的触发路径往往较深，通常需要多个不同维度的输入执行较深的逻辑交互才能触发，这是目前测试工具所不具备的能力。下面我们以 HDFS-13279 为例，详细介绍现有工具面临的主要挑战以及未来可能的解决方法。

典型案例：数据存储管理缺陷 HDFS-13279 是一个逻辑缺陷，由配置更新、用户请求负载和节点网络故障的测试输入共同作用，导致 HDFS 系统负载均衡分配错误，最终引发数据节点服务中断。图 13 展示了触发该缺陷的五个关键测试步骤及主要执行逻辑，图 14 展示了该缺陷及其修复的核心代码片段。

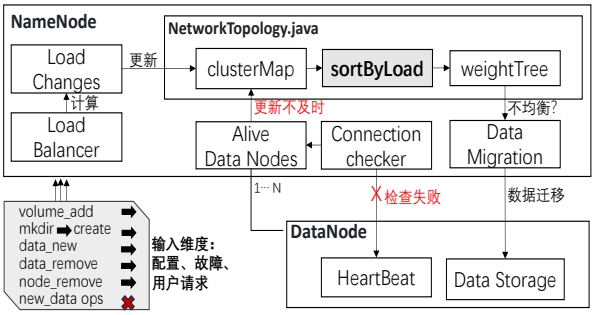


图 13 缺陷 HDFS-13279 的主要触发流程

```
1  private void sortByLoad(T[] nodes  ) {
2      /** Sort weights for the nodes array */
3      TreeMap<Integer, List<T>> weightedNodeTree =
4          new TreeMap<>();
5      int nWeight;
6
7      // Sort nodes which have the same weight using secondarySort.
8      for (List<T> l : weightedNodeTree.values()) {
9        Collections.shuffle(l);
10       for (T n : l) {
11         nodes[idx++] = n;
12       }
13     }
14+    Preconditions.checkState(idx != activeLen,
15+        "Sorted the wrong number of nodes!");
16 }
```

图 14 缺陷 HDFS-13279 及其修复的核心代码片段

在 HDFS 集群中，NameNode 负责存储并管理整个文件系统的元数据，包括文件和目录的结构、权限，以及每个文件块所在的 DataNode 信息。DataNode 则是负责实际数据存储和管理的关键组件。要触发这个缺陷，需要至少 5 个关键步骤：步骤①，通过配置动态更新指令挂载新卷，扩容某些 DataNode 存储空间从而激活负载均衡器 Load Balancer。步骤②，HDFS 接收到一系列数据新增存储请求后，Load Balancer 开始计算负载变化并相应更新存储分布。HDFS 使用一个 clusterMap 记录已连接的 DataNode，并使用一个 weightTree 对存储负载进行排序。步骤③，如果由于大负载数据删除变化请求导致树不平衡，就会触发数据迁移。步骤④，然而如果扩容的这个 DataNode 恰好在此时因网络故障或其他原因导致下线，就会触发这个数据迁移管理逻辑缺陷。由于断开的 DataNode 的状态没有及时在 clusterMap 中更新，系统错误地认为该 DataNode 仍然在线。因此，迁移的数据计算会出错，造成“热点”问题（某些节点的数据未被迁移出，仍然保留在原处）。步骤⑤，任何针对这些“热点”DataNode 的用户请求操作将会延缓甚至中断处理。这个逻辑缺陷会导致 HDFS 的关键数据服务挂起，从而影响 HDFS 的可用性。修复此缺陷的过程如图 14 的第 14-15 行所示，通过添加一个实时的状态检查器即可修复此缺陷。

案例启发：分布式系统由于逻辑复杂性，其代码实现过程中难免会出现缺陷。一个小的代码实现错误很可能会放大影响整个分布式系统，导致严重的后果。我们可以从这个案例中总结出一个重要的教训：一些分布式系统的代码实现错误往往隐藏在深层路径中，要触发它们，需要执行许多来自不同输入维度的操作，包括系统配置修改、用户负载请求以及故障注入。在这个案例中，至少需要 7 个步骤来产生足够频繁的负载变化，从而启动数据迁移过程，而在此过程中，DataNode 节点恰好由于各种故障而下线。因此，未来可以设计高效的多维度输入策略，以探索分布式系统的深层次交互逻辑，从而挖掘更多潜在的代码缺陷。

4.3 多维度输入测试初步实验

为了评估多维度输入策略对分布式系统动态测试的有效性，我们集成了上述四个维度的输入生成方法，挑选了每个输入维度上测试性能较好的工具作为集成基础对象（即系统配置生成工具 ECFuzz、用户请求负载生成工具 DUPTester、节点消息生成工具 Tyr 和故障注入工具 Chronos），形成了初步简单原型 MultiGen，其架构如图 15 所示。

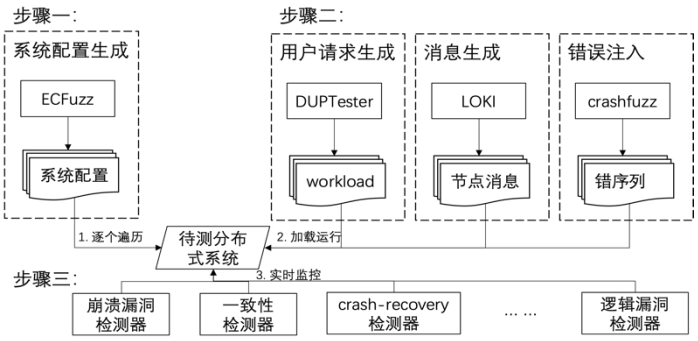


图 15 MultiGen 测试流程

MultiGen 的测试过程主要分为三个步骤。步骤一：首先，ECFuzz 基于待测分布式系统的默认配置文件，通过参数依赖感知的智能变异策略，对配置中的关键参数运行多轮模糊测试，生成大量系统配置测试输入文件，并进行配置有效性验证。根据 ECFuzz 的实验结果，该工具通常在 3-6 轮变异测试后测试过程趋于收敛。因此，MultiGen 仅选择前六轮生成的系统配置作为输入，逐个遍历加载经过验证后的系统配置文件，启动待测系统。步骤二：待测系统启动后，DUPTester、Tyr 以及 Chronos 分别并行生成三个不同维度的测试输入，直接作为输入由待测系统加载运行。步骤三：同时运行上述四个工具的缺陷检测器，实时监控待测分布式系统的状态，捕捉系统异常行为并输出系统缺陷。当待测系统加载完系统配置文件并启动系统后，步骤二、三不停迭代执行直到待测系统覆盖率收敛为止，结束步骤二、三。随后系统重新加载新的系统配置文件，重新执行步骤二、三的测试流程，直到用户中断或定时器到时。

表 6 MultiGen 在待测分布式系统上的测试表现

分布式系统	HDFS		ZooKeeper		IPFS	
测试覆盖	行(%)	分支	行(%)	分支	行(%)	分支
MultiGen	31.48%	43148	21.93%	16185	20.47%	2743
缺陷发现	崩溃缺陷	逻辑缺陷	崩溃缺陷	逻辑缺陷	崩溃缺陷	逻辑缺陷
MultiGen	5	7	5	5	3	4

表 6 展示了多维度测试输入工具 MultiGen 在三个待测分布式系统上运行 24 小时后的测试表现。与这四个典型先进动态测试工具相比，MultiGen 在三个待测分布式系统上总是能取得更好的代码行覆盖率和分支覆盖数。与测试工具 ECFuzz、 DUPTester、 Tyr 和 Chronos 相比，MultiGen 分别提升了 31%、 18%、 38% 和 34%的代码行覆盖率，能够多测试分支覆盖数 27%、 12%、 28% 和 23%。根据这些数据我们可以发现多维度输入策略能够有效提高测试效率，从而提升测试工具的代码覆盖情况。这是因为不同维度的输入之间存在依赖关系，例如同样的客户请求在不同系统配置下的代码执行路径可能不同（即执行了不同的代码分支）。通过将不同维度的输入简单集成，能够有效生成一些跨输入维度的测试场景，从而提高整体测试覆盖效果。

MultiGen 除了能覆盖原本测试工具发现的所有 21 个缺陷外，还在 HDFS、ZooKeeper 和 IPFS 系统上分别多找到 2、3、3 个系统缺陷，并且成功复现了缺陷 HDFS-13279。这表明不同测试维度输入的简单随机组合，也能有效覆盖一些深层次系统行为逻辑，从而揭露新的未知缺陷。

总体而言，尽管目前简单多维度集成工具 MultiGen 在测试效果上取得了一些进步，但对分布式系统的代码覆盖率仍然较低，挖掘的缺陷总数 29 仍与系统中的实际缺陷数量存在较大差距。一方面，这是因为分布式系统交互复杂，不同开发人员在开发各自模块时往往会采取防御性编程策略，从而存在许多无法被触发执行的“死代码”；另一方面，由于分布式系统的不同维度输入之间交互逻辑庞大且复杂，目前的动态测试工具无法生成足够复杂的测试场景来触发执行系统的深层代码路径。针对第一种情况，可以通过一些静态分析工具

并辅以人工分析，识别出这些“死代码”，在进行测试覆盖率统计时将其剔除。针对第二个问题，分布式系统的四个输入维度不是独立存在的，它们逻辑交织、交互依赖，共同作用于整个分布式系统，保证整体系统的可用性与安全性。目前 MultiGen 的多维度输入策略较为简单，但仍取得了不错的初步成效。未来的测试方案应该着眼于开发更高效且智能化的多维度测试输入方法，更加深入挖掘不同维度的逻辑依赖关系，并基于此分析建立详尽的测试场景模型，探索高效的多维度测试输入生成技术，以进一步优化测试效果。

5 总结与展望

5.1 分布式系统动态测试工作的总结

分布式系统在现代计算和存储密集型应用中发挥着重要作用，保障其安全性和可靠性至关重要，因此挖掘潜在的代码缺陷成为学术界和工业界的研究重点。近期也逐步涌现出大量分布式系统动态测试工具。其中，部分工具专注于分布式系统的逻辑缺陷，如 NEAT、Fluffy 和 DUPTester 等工具通过定义和探索逻辑问题的测试准则来检测缺陷；一些工具聚焦于安全漏洞检测，如 LOKI 用于内存安全检测，MPChecker 用于权限安全检测；另有工具专注于一致性缺陷检测，如 Tyr 的分布式共识一致性检测，CoFi 和 Modulo 的崩溃一致性检测，以及 FlyMC、TSVD 等工具的并发数据一致性检测。此外，IASO 和 Perseus 等工具专注于性能问题的测试，CrashFuzz 和 Phoenix 则针对鲁棒性问题进行测试，ScaleCheck 和 ChatT 等工具关注系统扩展性问题。这些动态测试技术在诸如 HDFS、ZooKeeper、IPFS 等流行的分布式系统上发现了大量缺陷，显著提升了系统的可用性、安全性和稳定性。

目前，分布式系统动态测试的局限性主要体现在以下四个方面：（1）**覆盖低**：分布式系统的代码规模庞大，包含复杂的行为逻辑。目前的动态测试工具覆盖率较低，许多深层次的代码逻辑未被测试，可能隐藏着大量缺陷。（2）**维度少**：多数动态测试工具仅关注一个或两个维度的输入，难以联动多个维度。这限制了测试的效率，因为一些潜藏的系统缺陷需要多维度的输入才能被触发（例如 5.2.3 小节中的缺陷 HDFS-13279）。（3）**适配差**：不同分布式系统的特性和输入语法差异较大，跨系统适配难度高，现有工具往往需要大量定制工作才能适用于不同类型的系统。此外，分布式系统的业务逻辑差异巨大，导致定义通用的缺陷检测器变得极为困难。目前，大多数工具集中在处理通用的缺陷类型，如性能下降、系统容错等。然而，涉及到更深层次的代码逻辑缺陷，如数据存储的准确性和任务分配的合理性等问题，往往被现有工具忽视。（4）**复现难**：分布式系统的环境复杂、状态多变，错误往往与多种历史交互相关，导致缺陷难以复现。完整复现错误需记录并重现所有节点的执行路径，这在实践中难以实现。

5.2 未来展望

未来的分布式系统动态测试技术可能有如下的发展方向。

（1）多维度输入协作生成技术。通过我们在第四章的初步实验，结果显示多维度输入能够显著提高测试覆盖率，进而挖掘出更多的系统缺陷。然而，目前的多维度输入集成方式仍然相对简单，尚未充分考虑不同维度之间的协作与交互关系。根据第二章提出的缺陷威胁模型，我们将分布式系统的输入划分为系统配置、用户输入、节点间消息与系统环境交互四个维度。然而，这些维度并非独立存在，而是彼此之间存在直接或间接的依赖关系。例如，在分布式文件系统中，文件存储模式通常有三种：DAS（块存储）、SAN（文件存储）和 NAS（对象存储）。这些存储模式由系统配置决定，并且不同的存储模式会影响输入生成策略。在对象存储模式下，输入样例应尽量覆盖更多复杂的对象关系；而在文件存储模式下，输入样例则应涵盖所有文件类型。同样，用户输入也会影响节点间的消息通信和系统环境交互。例如，频繁的大文件输入可能触发更多的节点间数据同步消息以及更多的磁盘 IO 操作。因此，未来的研究应重点探讨如何高效协同生成多维度测试输入。可以考虑通过精确构建系统各维度之间的输入关联模型，并基于该模型自动化生成大量高效的输入样例，最大限度地覆盖分布式系统的代码路径，进而发现更多潜在的系统缺陷。

（2）智能化通用测试输入合成。目前测试工具即使在相同应用领域内对不同的分布式系统进行测试，仍

需大量人力进行适配。以分布式文件系统为例，HDFS、GlusterFS 和 IPFS 输入语法格式相似，但将测试工具适配到新的系统仍需对其代码进行修改，以满足新系统的语法和语义要求。每个分布式文件系统都有其独特的设计特征和用户输入指令，仅使用共同的输入语法往往会遗漏许多系统特有的逻辑特征。跨领域的适配成本更大，需要投入更多的人力资源。为了实现自动化适配，未来可以设计统一的自动化工具，通过将不同建模描述语言（如 UML、ANTLR、JSON）或系统说明文档等定义的输入规范转化为统一的分布式输入标准模型，并基于该模型自动生成测试输入。此外，未来测试输入生成还可能结合以下技术：一是利用机器学习和数据挖掘技术，学习和模拟真实工作负载中的数据分布和用户输入，生成更真实、多样化的测试输入；二是使用符号执行技术和污点分析技术，生成更加复杂且能覆盖更深路径的测试输入；三是通过大语言模型技术，自动化适配不同类型分布式系统的测试输入，并生成能触发不同逻辑路径和边界情况的测试用例。

（3）丰富化缺陷准则定义。越丰富的缺陷检测器能够有效地识别出越多的系统缺陷。早期，大多数动态测试工具都聚焦于解决分布式系统的通用问题，如并发冲突、故障容错失效、性能下降等。随着动态测试在分布式领域的发展，越来越丰富的缺陷类型逐渐被定义，一些工具开始关注更细致领域的缺陷准则构建。例如，Tyr 和 Phoenix 等工具聚焦于区块链系统的去中心化共识逻辑缺陷；Chronos 探索了分布式系统中的超时处理错误缺陷；DUPTester 关注了分布式系统升级更新时的兼容逻辑问题；ScaleCheck 等工具定义了分布式系统的可扩展性缺陷。然而，目前仍然存在大量的分布式系统细分领域，其中的关键逻辑（例如负载均衡机制、数据备份机制等）缺乏精准的缺陷准则定义，导致许多逻辑缺陷依旧潜藏在代码中。未来，可以在分布式系统的细分功能领域中定义更丰富、精准的缺陷准则。

（4）自动化缺陷复现技术。缺陷自动化复现旨在重现分布式系统中的已知或潜在缺陷，以便进一步分析、修复和提升系统的安全性。这项技术在保障分布式系统的安全性和稳定性方面至关重要。然而，由于分布式系统的复杂性和逻辑多样性，动态测试中检测到的缺陷往往难以实现自动化复现。未来可能的研究方向包括以下三点：1. 基于快照的日志重放技术：在分布式系统运行时使用快照技术快速记录系统状态，包括内部状态、网络通信、数据状态等。结合缺陷发生前的日志信息，从最近的正常状态快照开始模拟日志行为，以复现缺陷并定位其根本原因。2. 基于关键行为插桩的复现技术：识别系统源码中的关键行为并进行插桩，记录测试阶段的信息。在复现阶段，这些信息用于控制逻辑流，精确重现缺陷场景。3. 持续开放共享的缺陷数据库与平台：建立一个开放共享的分布式系统缺陷数据库和平台，供研究人员共享缺陷信息、复现脚本和解决方案。通过历史缺陷的积累和复现过程，启发新的缺陷发现和复现，加速缺陷修复和系统安全性的提升。

References:

[1] W Ahmed and YW Wu. A survey on reliability in distributed systems[J]. Journal of Computer and System Sciences, 2013, 79(8): 1243-1255. [DOI: 10.1016/j.jcss.2013.02.006]

[2] AC Orgerie, MD Assuncao and L Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems[J]. ACM Computing Surveys, 2014, 46(4): 1-31. [DOI: 10.1145/253263]

[3] XL Zhang, JH Yang, XQ Sun, JP Wu. Survey of Geo-Distributed Cloud Research Progress[J]. Journal of Software, 2018, 29(7): 2116-2132(in Chinese). [DOI:10.13328/j.cnki.jos.005555]

[4] Apache ZooKeeper. Zookeeper security bugs in timeout machanisms. https://blog.cloudflare.com/october-2021-facebook-outage/.

[5] ZOOKEEPER-3189. Zookeeper connect timeout. https: //issues.apache.org/jira/browse/ZOOKEEPER-3189.

[6] A Legay, B Delahaye and S Bensalem. Statistical model checking: An overview[C]//International conference on runtime verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010: 122-135. [DOI: 10.1145/3555776.3577720]

[7] R Sinha, S Patil, L Gomes and V Vyatkin. A survey of static formal methods for building dependable industrial automation systems[J]. IEEE Transactions on Industrial Informatics, 2019, 15(7): 3772-3783. [DOI: 10.1109/TII.2019.2908665]

[8] MA Umar and C Zhanfang. A comparative study of dynamic software testing techniques[J]. International Journal of Advanced Networking and Applications, 2020, 12(3): 4575-4584. [DOI:10.35444/IJANA.2020.12301]

[9] Cimatti A, Clarke E, Giunchiglia F and Roveri M. NuSMV: A new symbolic model verifier[C]//Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. Springer Berlin Heidelberg, 1999: 495-499. [DOI: 10.1007/3-540-48683-6_44]

[10] M Kwiatkowska, G Norman and D Parker. PRISM: Probabilistic symbolic model checker[C]//International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002: 200-204. [DOI: 10.1007/3-540-46029-2_13]

[11] G Holzmann, E Najm and A Serhrouchni. SPIN model checking: An introduction[J]. International Journal on Software Tools for Technology Transfer, 2000, 2: 321-327. [DOI: 10.1007/s100090050039]

[12] Chen Jessica. On using static analysis in distributed system testing[C]//Engineering Distributed Objects: Second International Workshop, EDO 2000 Davis, CA, USA, November 2–3, 2000 Revised Papers. Springer Berlin Heidelberg, 2001: 145-162. [DOI: 10.1007/3-540-45254-0_13]

[13] V Lenarduzzi, N Saarimaki and D Taibi. On the diffuseness of code technical debt in java projects of the apache ecosystem[C]//2019 IEEE/ACM international conference on technical debt (TechDebt). IEEE, 2019: 98-107. [DOI: 10.1109/TechDebt.2019.00028]

[14] N Imtiaz, B Murphy and L Williams. How do developers act on static analysis alerts? an empirical study of coverity usage[C]//2019 IEEE 30th International Symposium on Software Reliability Engineering. IEEE, 2019: 323-333. [DOI: 10.1109/ISSRE.2019.00040]

[15] Fred J. Meyer and Dhirkt K. Pradhan. Dynamic testing strategy for distributed systems[J]. IEEE Transactions on Computers, 1989, 38(3): 356-365. [DOI: 10.1109/12.21122]

[16] A Ulrich and H König. Architectures for testing distributed systems[J]. Testing of Communicating Systems: Methods and Applications, 1999: 93-108. [DOI: 10.1007/978-0-387-35567-2_25]

[17] C Torens and L Ebrecht. Remotetest: A framework for testing distributed systems[C]//2010 Fifth International Conference on Software Engineering Advances. IEEE, 2010: 441-446. [DOI: 10.1109/ICSEA.2010.75]

[18] H Lefeuvre, VA Bădoiu, Y Chien, F Huici, N Dautenhahn and P Olivier. Assessing the Impact of Interface Vulnerabilities in Compartmentalized Software[M]//Proceedings of 30th Network and Distributed System Security. Internet Society, 2022. [DOI:10.14722/ndss.2023.24117]

[19] Han S, Shin K G, Rosenberg H A. Doctor: An integrated software fault injection environment for distributed real-time systems[C]//Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium. IEEE, 1995: 204-213.

[20] J Lu, C Liu, L Li, X Feng, F Tan, J Yang and L You. Crashtuner: Detecting crash-recovery bugs in cloud systems via meta-info analysis[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 114-130. [DOI: 10.1145/3341301.3359645]

[21] Y Chen, F Ma, Y Zhou, M Gu, Q Liao, Y Jiang. Chronos: Finding Timeout Bugs in Practical Distributed Systems by Deep-Priority Fuzzing with Transient Delay[C]//2024 IEEE Symposium on Security and Privacy. IEEE Computer Society, 2024: 109-109. [DOI: 10.1109/SP54263.2024.00109]

[22] L Qiang, C Li and C Haiming. Key Technologies and Applications of Internet of Things[J]. Computer Science, 2010, 37(6): 1-4. [DOI:10.3969/j.issn.1002-137X.2010.06.001]

[23] M Lahami and M Krichen. A survey on runtime testing of dynamically adaptable and distributed systems. Software Quality Journal, 2021, 29(2): 555-593. [DOI: 10.1007/s11219-021-09558-x]

[24] A Gosain and G Sharma. A survey of dynamic program analysis techniques and tools[C]//Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications. 2014: Volume 1. Springer International Publishing, 2015: 113-122. [DOI:10.1007/978-3-319-11933-5_13]

[25] Sternberg R J and Grigorenko E L. ALL TESTING IS DYNAMIC TESTING[J]. Issues in Education, 2001, 7(2).

[26] Fairley R E. Static analysis and dynamic testing of computer software[J]. Computer, 1978, 11(4): 14-23. [DOI: 10.1109/C-M.1978.218132]

[27] 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. Journal of Software, 2019, 30(1): 80-109(in Chinese). [DOI:10.13328/j.cnki.jos.005651]

[28] X Sun, R Cheng, J Chen, E Ang, O Legunsen and T Xu. Testing configuration changes in context to prevent production failures[C]//14th USENIX Symposium on Operating Systems Design and Implementation. 2020: 735-751.

[29] T Wang, Z Jia, S Li, S Zheng, Y Yu, E Xu, S Peng and X Liao. Understanding and detecting on-the-fly configuration bugs[C]//Proceedings of the 45th International Conference on Software Engineerin. 2023: 628-639. [DOI: 0.1109/ICSE48619.2023.00062]

[30] Q Chen, T Wang, O Legunsen, S Li, T Xu. Understanding and discovering software configuration dependencies in cloud and datacenter systems[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2020: 362-374. [DOI: 0.1145/3368089.3409727]

[31] A Hoffmann and B Neubauer. Deployment and configuration of distributed systems[C]//System Analysis and Modeling: 4th International SDL and MSC Workshop, SAM 2004, Ottawa, Canada, June 1-4, 2004. Springer Berlin Heidelberg, 2005: 1-16. [DOI: 10.1007/978-3-540-31810-1_1]

[32] Oppenheimer David. The importance of understanding distributed system configuration[C]//Proceedings of the 2003 Conference on Human Factors in Computer Systems workshop. 2003.

[33] LA Barroso, U Hölzle and P Ranganathan. The datacenter as a computer: Designing warehouse-scale machines. Springer Nature, 2019:189. [DOI: 10.5555/3306658]

[34] Shieber, J. Facebook outage. https://techcrunch.com/2019/03/14/facebook-blames-a-misconfigured-server-for-yesterdays-outage/.

[35] HS Gunawi, M Hao, RO Suminto, A Laksono, AD Satria, J Adityatama and KJ Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages[C]//Proceedings of the Seventh ACM Symposium on Cloud Computing. 2016: 1-16. [DOI: 10.1145/2987550.2987583]

[36] H Liu, S Lu, M Musuvathi, S Nath. What bugs cause production cloud incidents?[C]//Proceedings of the Workshop on Hot Topics in Operating Systems. 2019: 155-162. [DOI: 10.1145/3317550.3321438]

[37] Nagaraja, K., Oliveira, F., Bianchini, R., Martin, R. P., and Nguyen, T. D. Understanding and Dealing with Operator Mistakes in Internet Services[C]// 6th Symposium on Operating Systems Design and Implementation. 2004, 4: 61-76.

[38] Oppenheimer D, Ganapathi A and Patterson D A. Why do Internet services fail, and what can be done about it?[C]//4th Usenix Symposium on Internet Technologies and Systems. 2003. [DOI: 0.5555/1251460.1251461]

[39] Yuan, D., Luo, Y., Zhuang, X., Rodrigues, G., Zhao, X., Zhang, Y., Jain, P. U., and Stumm, M. Simple testing can prevent most critical failures: An analysis of production failures in distributed {Data-Intensive} systems[C]//11th USENIX Symposium on Operating Systems Design and Implementation. 2014: 249-265. [DOI: 10.5555/2685048.2685068]

[40] Hale B. Why every it practitioner should care about network change and configuration management[J]. Hale B., SolarWinds, 2012.

[41] DDOS Attack. Famous DDos attacks in history. https://www.cloudflare.com/zh-cn/learning/ddos/famous-ddos-attacks/.

[42] Huang C F, Tseng Y C, Wu H L. Distributed protocols for ensuring both coverage and connectivity of a wireless sensor network[J]. ACM Transactions on Sensor Networks, 2007, 3(1): 5-es. [DOI: 10.1145/1210669.1210674]

[43] P Porambage, C Schmitt, P Kumar, A Gurtov and M Ylianttila. Two-phase authentication protocol for wireless sensor networks in distributed IoT applications[C]//2014 IEEE Wireless Communications and Networking Conference. 2014: 2728-2733.

[44] W Zhong, C Yang, W Liang, J Cai, L Chen, J Liao and N Xiong. Byzantine fault-tolerant consensus algorithms: A survey[J]. Electronics, 2023, 12(18): 3801. [DOI: 10.3390/electronics12183801]

[45] Distler Tobias. Byzantine fault-tolerant state-machine replication from a systems perspective[J]. ACM Computing Surveys (CSUR), 2021, 54(1): 1-38. [DOI: 10.1145/3436728]

[46] M Abd-El-Malek, GR Ganger, GR Goodson, MK Reiter and JJ Wylie. Fault-scalable Byzantine fault-tolerant services[J]. ACM SIGOPS Operating Systems Review, 2005, 39(5): 59-74. [DOI: 10.1145/1095809.1095817]

[47] CVE-2021-39137. https://cve.mitre.org/cgi-bin/cvename.cgi?name= CVE-2021-39137.

[48] CVE-2020-26265. https://cve.mitre.org/cgi-bin/cvename.cgi?name= CVE-2020-26265.

[49] Correa da Silva and Flavio Soares. Knowledge-based interaction protocols for intelligent interactive environments[J]. Knowledge and Information Systems, 2013, 34: 219-242. [DOI: 10.1007/s10115-011-0464-7]

[50] Amazon Simple Storage Service (S3) team. Summary of the Amazon S3 Service Disruption. https://aws.amazon.com/cn/message/41926.

[51] D Wang, W Dou, Y Gao, C Wu, J Wei and T Huang. Model checking guided testing for distributed systems[C]//Proceedings of the Eighteenth European Conference on Computer Systems. 2023: 127-143. [DOI: 10.1145/3552326.3587442]

[52] Lamport Leslie. Specifying systems: the TLA+ language and tools for hardware and software engineers. in Computer, vol. 35, no. 9, pp. 81-81, Sept. 2002. [DOI: 10.1109/MC.2002.1033032]

[53] H Liu, G Li, JF Lukman, J Li, S Lu, HS Gunawi and C Tian. Dcatch: Automatically detecting distributed concurrency bugs in cloud systems[J]. ACM SIGARCH Computer Architecture News, 2017, 45(1): 677-691. [DOI: 10.1145/3037697.3037735]

[54] Michael Eddington. Peach Fuzzer: Discover Unknown Vulnerabilities. https://peachtech.gitlab.io/peach-fuzzer-community/.

[55] DBSCAN. Density-based spatial clustering of applications with noise. https://en.wikipedia.org/wiki/DBSCAN.

[56] Alibaba. ChaosBlade. https://github.com/chaosblade-io/chaosblade.

[57] Swagger. API Testing. https://swagger.io/solutions/api-testing/.

[58] J Yin, X Lu, X Zhao, H Chen and X Liu. BURSE: A bursty and self-similar workload generator for cloud computing[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 26(3): 668-680. [DOI: 10.1109/TPDS.2014.2315204]

[59] Netflix. Chaos monkey. https://netflix.github.io/chaosmonkey/.

[60] R Bodnarchuk and R Bunt. A synthetic workload model for a distributed system file server[C]//Proceedings of the 1991 ACM SIGMETRICS conference on Measurement and modeling of computer systems. 1991: 50-59. [DOI: 10.1145/107971.107978]

[61] Carroll J J, Anand P and Guo D. Preproduction deploys: Cloud-native integration testing[C]//2021 IEEE Cloud Summit. IEEE, 2021: 41-48. [DOI:10.1109/IEEECloudSummit52029.2021.00015]

[62] Pact. Contract Testing. https://docs.pact.io/.

[63] Halili E H. Apache JMeter. https://jmeter.apache.org/.

[64] J Yun, F Rustamov, J Kim and Y Shin. Fuzzing of embedded systems: A survey[J]. ACM Computing Surveys, 2022, 55(7): 1-33. [DOI: 10.1145/3538644]

[65] Y Zhang, J Yang, Z Jin, U Sethi, K Rodrigues, S Lu and D Yuan. Understanding and detecting software upgrade failures in distributed systems[C]//Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles. 2021: 116-131. [DOI: 10.1145/3477132.3483577]

[66] F Ma, Y Chen, M Ren, Y Zhou, Y Jiang, T Chen, H Li and J Sun. LOKI: State-Aware Fuzzing Framework for the Implementation of Blockchain Consensus Protocols[C]//Proceedings 2023 Network and Distributed System Security Symposium. 2023. [DOI:10.14722/ndss.2023.24078]

[67] Y Gao, W Dou, D Wang, W Feng, J Wei, H Zhong, T Huang. Coverage guided fault injection for cloud systems[C]// 2023 IEEE/ACM 45th International Conference on Software Engineering. 2023: 2211-2223. [DOI: 10.1109/ICSE48619.2023.00186]

[68] JaCoCo code coverage tool. https://www.eclemma.org/jacoco/.

[69] Googletest coverage. https://github.com/google/googletest.

[70] J Li, S Li, K Li, F Luo, H Yu, S Li and X Li. ECFuzz: Effective Configuration Fuzzing for Large-Scale Systems[C]//Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024: 1-12. [DOI: 10.1145/3597503.3623315]

[71] Zimmerer P. Test architectures for testing distributed systems[J]. 12th International software quality week, 1999.

[72] D Gupta, KV Vishwanath, M McNett, A Vahdat, K Yocum, A Snoeren and GM Voelker. DieCast: Testing distributed systems with an accurate scale model[J]. ACM Transactions on Computer Systems, 2011, 29(2): 1-48. [DOI: 10.1145/1963559.1963560]

[73] Y Wang, M Kapritsos, L Schmidt and M Dahlin. Exalt: Empowering Researchers to Evaluate {Large-Scale} Storage Systems[C]//11th USENIX Symposium on Networked Systems Design and Implementation. 2014: 129-141. [DOI: 10.5555/2616448.2616461]

[74] Stuardo C A, and Leesatapornwongsa T. {ScaleCheck}: A {Single-Machine} Approach for Discovering Scalability Bugs in Large Distributed Systems[C]//17th USENIX Conference on File and Storage Technologies. 2019: 359-373. [DOI: 10.5555/3323298.3323332]

[75] Hsaini, Sara, Salma Azzouzi, and My El Hassan Charaf. A temporal based approach for MapReduce distributed testing[J]. International Journal of Parallel, Emergent and Distributed Systems, 2021, 36(4): 293-311. [DOI: 10.1080/17445760.2021.1879068]

[76] W Li, S Li, X Liao, X Xu, S Zhou and Z Jia. Conftest: Generating comprehensive misconfiguration for system reaction ability evaluation[C]//Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering. 2017: 88-97. [DOI: 10.1145/3084226.3084244]

[77] AA Pranata, O Barais, J Bourcier and L Noirie. ChaT: Evaluation of Reconfigurable Distributed Network Systems Using Metamorphic Testing[C]//2021 IEEE Global Communications Conference. 2021: 1-6. [DOI: 10.1109/GLOBECOM46510.2021.9685879]

[78] R Lu, E Xu, Y Zhang, F Zhu, Z Zhu, M Wang, Z Zhu, G Xue, J Shu, M Li and J Wu. Perseus: A {Fail-Slow} Detection Framework for Cloud Storage Systems[C]//21st USENIX Conference on File and Storage Technologies . 2023: 49-64. [DOI: 10.5555/3585938.3585942]

[79] Y Chen, F Ma, Y Zhou, Y Jiang, T Chen and J Sun. Tyr: Finding consensus failure bugs in blockchain system with behaviour divergent model[C]//2023 IEEE Symposium on Security and Privacy. 2023: 2517-2532. [DOI: 10.1109/SP46215.2023.10179386]

[80] F Ma, Y Chen, Y Zhou, J Sun, Z Su, Y Jiang, J Sun and H Li. Phoenix: Detect and locate resilience issues in blockchain via context-sensitive chaos[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023: 1182-1196. [DOI: 10.1145/3576915.3623071]

[81] F Nikolaidis, A Chazapis, M Marazakis and A Bilas. Frisbee: automated testing of Cloud-native applications in Kubernetes[J]. arXiv preprint arXiv:2109. 2021, 10727. [DOI: 10.48550/arXiv.2109.10727]

[82] N Machado, J Pereira, FA Coelho, F Neves and F Maia. Minha: Large-scale distributed systems testing made practical[J]. 2020.

[83] Alberto Núñez, Pablo C. Cañizares, Manuel Núñez and Robert M. Hierons. TEA-Cloud: A formal framework for testing cloud computing systems[J]. IEEE Transactions on Reliability, 2020, 70(1): 261-284. [DOI: 10.1109/TR.2020.3011512]

[84] Yuan X and Yang J. Effective concurrency testing for distributed systems[C]//Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. 2020: 1141-1156. [DOI: 10.1145/3373376.3378484]

[85] Yang Y, Kim T and Chun B G. Finding consensus bugs in ethereum via multi-transaction differential fuzzing[C]//15th USENIX Symposium on Operating Systems Design and Implementation. 2021: 349-365.

[86] J Lu, H Li, C Liu, L Li and K Cheng. Detecting Missing-Permission-Check Vulnerabilities in Distributed Cloud Systems[C]//Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. 2022: 2145-2158. [DOI: 10.1145/3548606.3560589]

[87] Panasas. https://en.wikipedia.org/wiki/Panasas.

[88] RUBiS. https://en.wikipedia.org/wiki/Rubis_(company).

[89] BK Ozkan, R Majumdar, F Niksic, MT Befrouei and G Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees[J]. Proceedings of the ACM on Programming Languages, 2018, 2: 1-28. [DOI: 10.1145/3276530]

[90] R Meng, G Pîrlea, A Roychoudhury and I Sergey. Greybox Fuzzing of Distributed Systems[C]//Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. 2023: 1615-1629. [DOI: 10.1145/3576915.3623097]

[91] Jepsen. Distributed systems safety research. http://jepsen.io/.

[92] Kim B H, Kim T and Lie D. Modulo: Finding Convergence Failure Bugs in Distributed Systems with Divergence Resync Models[C]//USENIX Annual Technical Conference. 2022: 383-398.

[93] H Chen, W Dou, D Wang and F Qin. CoFI: Consistency-guided fault injection for cloud systems[C]//Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. 2020: 536-547. [DOI: 10.1145/3324884.3416548]

[94] JF Lukman, H Ke, CA Stuardo, RO Suminto and DH Kurniawan. Flymc: Highly scalable testing of complex interleavings in distributed systems[C]//Proceedings of the Fourteenth EuroSys Conference 2019. 2019: 1-16. [DOI: 10.1145/3302424.3303986]

[95] K Serebryany, D Bruening, A Potapenko and D Vyukov. {AddressSanitizer}: A fast address sanity checker[C]//2012 USENIX annual technical conference. 2012: 309-318. [DOI: 10.5555/2342821.2342849]

[96] T Leesatapornwongsa, M Hao, P Joshi, JF Lukman and HS Gunawi. {SAMC}:{Semantic-Aware} Model Checking for Fast Discovery of Deep Bugs in Cloud Systems[C]//11th USENIX Symposium on Operating Systems Design and Implementation . 2014: 399-414.

[97] B Panda, D Srinivasan, H Ke, K Gupta, V Khot and HS Gunawi. {IASO}: A {Fail-Slow} Detection and Mitigation Framework for Distributed Storage Services[C]//2019 USENIX Annual Technical Conference. 2019: 47-62. [DOI: 10.5555/3358807.3358812]

[98] Wang T, Jia Z, Li S, et al. Understanding and detecting on-the-fly configuration bugs[C]//2023 IEEE/ACM 45th International Conference on Software Engineering. 2023: 628-639. [DOI: 10.1109/ICSE48619.2023.00062]

[99] P Joshi, HS Gunawi and K Sen. PREFAIL: A programmable tool for multiple-failure injection[C]// 2011 ACM international conference on Object oriented programming systems languages and applications. 2011:171-188. [DOI: 10.1145/2048066.2048082]

[100] Gcov/Lcov. A Test Coverage Program from the GNU Project. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

[101] HS Gunawi, T Do, P Joshi and P Alvaro. {FATE} and {DESTINI}: A framework for cloud recovery testing[C]//8th USENIX Symposium on Networked Systems Design and Implementation. 2011. [DOI: 10.5555/1972457.1972482]

[102] P Deligiannis, AF Donaldson, J Ketema, A Lal and P Thomson. Asynchronous programming, analysis and testing with state machines[C]//Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2015: 154-164. [DOI: 10.1145/2737924.2737996]

[103] J Yang, T Chen, M Wu, Z Xu, X Liu, H Lin, M Yang, F Long, L Zhang and L Zhou. MODIST: Transparent model checking of unmodified distributed systems[C]// 6th USENIX Symposium on Networked Systems Design and Implementation. 2009: 213-228.

[104] K Kallas, F Niksic, C Stanford and R Alur. DiffStream: differential output testing for stream processing programs[J]. Proceedings of the ACM on Programming Languages, 2020, 4: 1-29. [DOI: 10.1145/3428221]

[105] Yang Y, Kim T and Chun B G. Finding consensus bugs in ethereum via multi-transaction differential fuzzing[C]//15th USENIX Symposium on Operating Systems Design and Implementation. 2021: 349-365. [DOI: DOI:10.1109/SP46215.2023.10179386]

[106] A Alquraan, H Takruri, M Alfatafta and S Al-Kiswany. An analysis of {Network-Partitioning} failures in cloud systems[C]//13th USENIX Symposium on Operating Systems Design and Implementation. 2018: 51-68. [DOI: 10.5555/3291168.3291173]

[107] P Deligiannis, M McCutchen, P Thomson, S Chen, AF Donaldson, J Erickson and C Huang. Uncovering Bugs in Distributed Storage Systems during Testing (Not in {Production!})[C]//14th USENIX Conference on File and Storage Technologies. 2016: 249-262. [DOI: 10.5555/2930583.2930602]

附中文参考文献:

[3] 张晓丽,杨家海,孙晓晴,吴建平.分布式云的研究进展综述.软件学报,2018,29(7):2116-2132. [DOI:10.13328/j.cnki.jos.005555]

[22] 刘强,崔莉,陈海明.物联网关键技术与应用[J].计算机科学,2010,37(6):1-4. [DOI:10.3969/j.issn.1002-137X.2010.06.001]

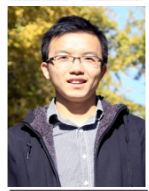
[27] 张健,张超,玄跻峰,熊英飞,王千祥,梁彬,李炼,窦文生,陈振邦,陈立前,蔡彦.程序分析研究进展.软件学报,2019,30(1):80-109. [DOI:10.13328/j.cnki.jos.005651]

陈元亮(1995-),男,博士,CCF 学生会员,主要研究领域为漏洞挖掘与软件安全分析.

马福辰(1997-),男,博士,CCF 学生会员，主要研究领域为区块链、分布式系统安全分析.

周远航(1999－),女,硕士,CCF 学生会员,主要研究领域为密码库，区块链系统安全分析.

颜臻(2001－),男,硕士,CCF 学生会员,主要研究区块链与分布式系统安全分析.



姜宇(1989－),男,博士,副教授,博士生导师,CCF 专业会员,主要研究领域为软件工程,物理信息融合系统.

孙家广(1946－),男,教授,博士生导师，主要研究领域为计算机图形学，计算机辅助设计，软件工程与系统.