软件学报 ISSN 1000-9825, CODEN RUXUEW
2024,35(12):5363−5381 [doi: 10.13328/j.cnki.jos.007066] [CSTR: 32375.14.jos.007066]

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

基于 TLA+形式化规约的 Raft 协议测试*

王 栋[1,2], 窦文生[1,2], 高 钰[1,2], 吴陈傲[1,2], 魏 峻[1,2], 黄 涛[1,2]

[1](计算机科学国家重点实验室 (中国科学院 软件研究所), 北京 100190)

[2](中国科学院大学, 北京 100049)

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

摘　要: Raft 是最为流行的分布式共识协议之一. 自 2014 年被提出以来, Raft 协议及其变体在各种分布式系统中被广泛应用. 为了证明 Raft 协议的正确性, 开发者使用 TLA+形式化规约对协议设计进行了建模和验证. 但由于抽象的形式化规约与实际的系统实现源码间存在鸿沟, 基于 Raft 实现的分布式系统中仍然会违背协议设计并引入复杂的缺陷. 设计基于 TLA+形式化规约的测试方法来检测 Raft 协议实现中的缺陷. 具体而言, 将形式化规约匹配到相应的系统实现, 并用形式化规约所定义的状态空间来指导系统实现的测试过程. 为评估所提方法的可行性和有效性, 针对两个不同的 Raft 实现进行系统化测试, 并发现 3 个未知缺陷.

关键词: Raft; 分布式系统; 软件测试; 模型检查

中图法分类号: TP311

中文引用格式: 王栋, 窦文生, 高钰, 吴陈傲, 魏峻, 黄涛. 基于TLA+形式化规约的Raft协议测试. 软件学报, 2024, 35(12): 5363–5381. http://www.jos.org.cn/1000-9825/7066.htm

英文引用格式: Wang D, Dou WS, Gao Y, Wu CA, Wei J, Huang T. Raft Protocol Testing Based on TLA+ Formal Specification. Ruan Jian Xue Bao/Journal of Software, 2024, 35(12): 5363–5381 (in Chinese). http://www.jos.org.cn/1000-9825/7066.htm

Raft Protocol Testing Based on TLA+ Formal Specification

WANG Dong[1,2], DOU Wen-Sheng[1,2], GAO Yu[1,2], WU Chen-Ao[1,2], WEI Jun[1,2], HUANG Tao[1,2]

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

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

Abstract: Raft is one of the most popular distributed consensus protocols. Since it was proposed in 2014, Raft and its variants have been widely used in different kinds of distributed systems. To prove the correctness of the Raft protocol, developers use the TLA+ formal specification to model and verify its design. However, due to the gap between the abstract formal specification and practical implementation, distributed systems that implement the Raft protocol can still violate the protocol design and introduce intricate bugs. This study proposes a novel testing technique based on TLA+ formal specification to unearth bugs in Raft implementations. To be specific, the study maps the formal specification to the corresponding system implementation and then uses the specification-defined state space to guide the testing in the implementations. To evaluate the feasibility and effectiveness of the proposed approach, the study applies it on two different Raft implementations and finds 3 previously unknown bugs.

Key words: Raft; distributed system; software testing; model checking

分布式共识协议是分布式计算领域的基础协议之一, 在各类分布式系统如分布式数据库[1,2]、分布式协调系统[3,4]、区块链系统[5,6]中被广泛应用. 与其他分布式共识协议如 Paxos[7]相比, Raft 协议[8]由于其易理解和易实现的特性, 成为最为流行的分布式共识协议之一. Raft 协议包含领导者选举、日志复制、日志精简、成员变更等多个场景, 能够为用户提供读写日志的服务, 能够容忍多种故障如节点重启、消息丢失等. 其被应用在多个分布式系

* 基金项目: 国家自然科学基金 (62072444, 62302493); 国家自然科学基金联合基金 (U20A6003)
收稿时间: 2022-12-12; 修改时间: 2023-06-12; 采用时间: 2023-10-10; jos 在线出版时间: 2024-01-31
CNKI 网络首发时间: 2024-02-02

统中如 CockroachDB[2]、TiDB[1]中. 除此以外, Raft 还有很多不同的变体[9−11], 这些变体针对不同的需求如性能优化对 Raft 进行了一定程度上的修改.

为了证明复杂分布式协议的正确性, 近年来研究者和开发者常使用形式化方法对协议设计进行验证. 其中, 最为广泛使用的是利用形式化语言 TLA+[12]对分布式协议进行建模, 并进一步利用模型检查器 TLC 对 TLA+规约中定义的性质进行验证. 例如, Ongaro 利用 TLA+对 Raft 中的领导者选举和日志复制场景进行了验证[13]. 亚马逊公司和微软公司的开发者利用 TLA+分别对 DynamoDB 和 Azure 等分布式系统的关键协议进行了建模与验证, 并成功发现若干设计上的缺陷[14,15]. 南京大学的谷晓松等人使用 TLA+对 Raft 协议的变体 ParallelRaft 进行了验证并发现了“幽灵日志”问题[16]. 但是, 即使利用 TLA+形式化规约对协议设计的正确性进行了验证, 开发者也不能保证协议在实现中是没有任何缺陷的[17]. 为此, 研究者们提出了一系列的框架[18,19]来验证分布式协议在实现层面的正确性. 然而以这些方式来进行验证十分复杂且代价极大. 验证一个相对简单的分布式协议实现通常需要多人年的开发代价[20]. 因此, 将这些验证框架应用于现实世界中的分布式协议仍具备相当的困难.

实现级模型检查器[21−25]是一类专用于在分布式系统实现中发现缺陷的测试技术. 该技术通过在分布式系统运行时对系统行为如消息发送和接收进行拦截, 并控制行为的执行顺序, 以使系统进入不同的状态. 实现级模型检查器使用用户提供的断言对分布式系统进行检查, 若系统的运行时行为或状态违反断言, 则报告为系统实现中的缺陷. 实现级模型检查器主要面临两类问题. 第一, 测试预言 (Oracle) 有效性的问题. 实现级模型检查器依赖于开发者提供的测试断言来判断测试过程中是否出现了缺陷. 例如, SAMC[24]要求开发者为每一个被检查的属性编写一个可运行脚本, 以供 SAMC 在测试过程中检查属性是否被违反. 而手动编写的测试断言难以覆盖分布式系统的所有状态及行为. 因此, 实现级模型检查器的缺陷检测能力极大地受到了测试断言的限制. 第二, 路径爆炸问题. 由于实现级模型检查器无法预知每一个系统行为所导致的状态. 因此, 在测试过程中, 每当有新的系统行为出现时, 实现级模型检查器都需要对该行为进行测试. 而在分布式系统庞大的状态空间中, 存在海量的重复行为, 如带有相同参数的用户请求. 实现级模型检查器会对这些行为进行重复测试, 进而导致需要测试的状态路径数量呈爆炸性增长.

本文中, 我们提出了一种新的测试方法来对基于 Raft 协议实现的分布式系统进行系统化测试. 对于给定分布式系统的源码及相应的 TLA+形式化规约, 该方法能够验证该实现是否遵从了形式化规约所定义的 Raft 协议设计. 具体地, 我们将 TLA+规约中的各类要素映射到协议实现的代码中, 并且利用规约中定义的状态空间来引导测试过程. 在测试中, 我们控制被测系统的各种不确定性, 以确保系统运行能够遵照状态空间中的状态变换. 最后, 我们对系统在测试中的行为及状态进行监控, 并将其与状态空间中的状态变化进行比较, 任何行为及状态值的差异都表示在系统实现中出现了一个违背 Raft 协议设计的缺陷. 为了评估该方法的可行性和有效性, 我们将其应用在了两个基于 Java 实现的 Raft 系统 Xraft[11]、Raft-Java[10]上. 最终, 我们成功在两个分布式系统中发现了 5 个缺陷. 其中 3 个缺陷被确认为新的缺陷, 剩余 2 个为已知缺陷. 除此以外, 我们还发现了两个存在于 Raft 的 TLA+规约[13]中的缺陷.

本文第 1 节介绍 Raft 协议、TLA+形式化规约以及 TLC 模型检查的相关知识与背景. 第 2 节介绍基于 TLA+形式化规约的 Raft 协议测试及实现. 第 3 节通过对测试方法所发现缺陷的讨论验证了测试方法的有效性. 第 4 节介绍我们讨论该方法的局限性、我们在测试中学习到的经验以及未来的研究方向. 第 5 节总结前文未涉及的相关工作. 最后总结全文.

1 基础知识

1.1 Raft 协议及其实现

Raft 是一个经典的分布式共识协议. 其包含 4 个使用场景, 即领导者选举、日志复制、快照、成员变更. 除此之外, Raft 还能够容忍各类系统故障, 即通信超时、节点失效、节点重启、消息丢失、消息重复.

领导者选举 (leader election) 场景开始于某个节点发现其与当前领导者节点的通信发生超时故障时. 该节点会成为候选者, 并向其他所有节点发送 *RequestVote* 消息以请求其他节点投票. 如果集群中的大多数节点 (Quorum)

认可了该投票请求, 则该节点成为领导者, 其他节点成为跟随者.

日志复制 (log replication) 场景可以在有领导者被选出时运行. 当领导者节点收到客户端发起的日志写请求时, 领导者发送 *AppendEntries* 消息至其他跟随者节点. 跟随者检查该请求是否合法并回复领导者. 当领导者收到集群中大多数节点的确认请求后, 则将该请求中的操作在其状态机中执行并回复给客户端.

日志精简 (log compaction) 场景被用来节省存储和网络资源. 由于 Raft 存储的数据是一个追加列表, 当整个 Raft 集群执行了大量的日志写操作后, 其存储的日志数据会占据相当大的存储资源. 当有新的成员节点加入集群并进行日志复制时, 同样也会消耗大量的网络资源. 因此, Raft 提供了快照功能, 可以由客户端或自动周期性地发起, 领导者节点会将 *InstallSnapshot* 消息发送给所有的跟随者节点. 收到消息并产生快照的节点会将最新的数据持久化到硬盘中, 并将过期的日志进行回收.

成员变更 (membership change) 场景被用来提高系统的可用性, 即集群所提供的服务在节点配置变更期间仍然可用. 为了达到这个目的, Raft 设计了一个瞬时联合共识 (transactional joint consensus) 阶段. 当有新的节点加入时, 整个集群会进入该阶段, 并允许两个不同的 Quorum 存在, 节点间会互相复制成员变更日志. 直到所有节点再次达成共识, Raft 会退出该阶段.

1.2 TLA+形式化规约

TLA+[12]是一种由 Lamport 开发的基于时序逻辑的形式化规约语言, 被广泛用于对分布式协议进行建模. 作为一种面向状态的建模技术, TLA+可以将目标程序抽象为一个状态机. 在 TLA+形式化规约中, 开发者使用 3 类要素即变量 (variable)、动作 (action)、常量 (constant) 来定义整个状态空间; 使用属性 (property) 来定义状态空间需要满足的约束. 下面, 我们分别对这些要素进行介绍.

变量代表了系统状态的具体值. 变量值会在对 TLA+规约进行验证的过程中被更改, 以生成不同的状态. 开发者在 TLA+代码中使用关键字 *VARIABLES* 来声明变量.

动作描述了状态之间的变化逻辑. 在 TLA+规约中, 动作以方法 (function) 的形式实现, 但并非所有的方法都是动作. 仅在关键字 Next 后、以析取 (∨) 联结符相连的方法为动作.

常量被用来表示 TLA+规约中的特殊值或约束状态空间大小. 开发者使用关键字 *CONSTANTS* 来声明常量, 并在对规约进行模型检查前对常量进行赋值.

属性定义了状态值需要满足的约束. 属性对于分布式协议的状态空间构建没有帮助. 因此, 本文介绍的利用 TLA+形式化规约定义的状态空间对测试过程进行指导的方法对于属性并不做特殊考虑.

在本文中, 我们主要使用 Raft 作者所开发的 TLA+规约[13]. 该规约共包含 468 行 TLA+代码, 仅对领导者选举及日志复制场景进行了建模. 我们补充了 51 行 TLA+代码以实现日志精简场景.

1.3 使用 TLC 模型检查器对 TLA+形式化规约进行验证

TLC[26]是最常用的针对 TLA+形式化规约的模型检查器. 当进行模型检查时, 开发者首先对所有常量进行赋值, 并指定需要检查的属性 (可选). 然后, 模型检查器会从规约中所定义的初始状态 (*init*) 出发, 通过执行不同的动作, 枚举所有的变量值, 以生成所有可能的状态. 由于使用了常量对分布式协议的状态空间进行了限制, TLC 所检查的状态是有限的. 整个模型检查过程会在两种情况下终止, 即发现违背属性的状态或所有状态都被检查并没有违背属性. 如果开发者没有指定任何属性, 则 TLC 仅会生成状态而不会进行状态检查. TLC 最终可以将整个状态空间输出成为一张有向图. 图中的点代表拥有不同值的不同状态, 边代表产生状态的各个动作.

通过对 TLA+形式化规约进行模型检查, 我们可以找出系统设计上违背系统属性的反例. 但验证过的分布式系统依然能够在实现中存在缺陷, 且这些缺陷都不能被以模型检查为代表的形式化方法所发现[17].

2 基于 TLA+形式化规约的 Raft 协议测试

我们提出基于 TLA+形式化规约的测试方法来对分布式系统实现进行测试. 该方法相比现有测试方法有以下 3 个优势. 首先, TLA+形式化规约比分布式系统实现更为抽象, 相应地所定义的状态空间更小. 其次, 通过对规约

进行模型检查, 开发者可以预知每一个动作的预期状态, 因此不需要对系统行为进行大量重复测试. 最后, 模型检查所产生的状态空间中有确定的动作序列以及每一个动作所导致的状态, 可以作为分布式系统测试的预言, 而无需再由开发者提供断言脚本来判断是否出现了缺陷.

但是, 基于 TLA+形式化规约对分布式系统进行测试面临着 3 个主要挑战. 第一, TLA+规约与系统实现之间存在鸿沟. TLA+形式化规约代码基于抽象的时序逻辑进行开发, 而系统实现以可执行代码如 Java 进行开发. 如何将 TLA+规约的代码与实际的系统源码相匹配是一个挑战. 第二, 对 TLA+规约进行模型检查后所生成的状态空间是抽象的、由离散的动作组成的图, 然而分布式系统的状态空间是具体的、连续的行为序列. 如何用抽象的状态空间去生成能在目标系统实际运行的测试用例是一个挑战. 第三, 分布式系统在运行时充满不确定性. 如何在测试中控制不确定性, 确保目标系统遵从测试用例运行是一个挑战.

针对第 1 个挑战, 我们设计了一个方法将 TLA+形式化规约中的元素映射到分布式系统实现源码中 (第 2.2 节). 针对第 2 个挑战, 我们对状态空间图进行遍历并生成带有动作及预期状态的路径, 作为目标系统的测试用例 (第 2.3 节). 针对第 3 个挑战, 我们基于 TLA+规约元素的映射结果, 在系统源码中各种系统动作的首尾进行插桩, 以控制动作的发生序列, 并对实际状态进行检查 (第 2.4 节).

在本节中, 我们首先简要介绍该方法的工作流程, 随后详细介绍其中的 3 个主要步骤. 最后, 我们介绍该方法的实现细节.

2.1 方法概述

图 1 展示了基于 TLA+形式化规约的测试方法的概述. 整个工作流程包括 3 个主要阶段: 将 TLA+规约映射到实现代码, 测试用例生成以及基于测试用例进行的受控测试.

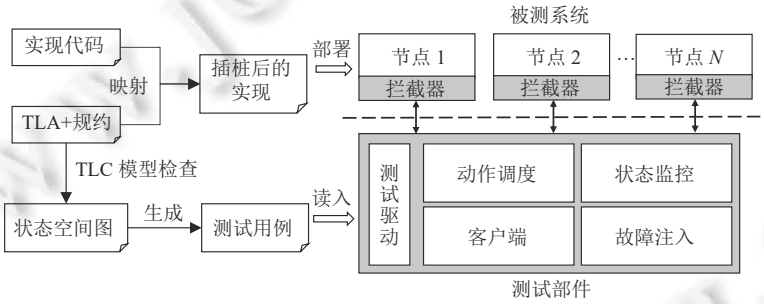


图 1 基于 TLA+形式化规约的测试方法概述

我们以被测系统实现源码以及相应的 TLA+形式化规约作为输入. 首先利用 TLC 对 TLA+规约进行模型检查, 并输出状态空间图. 然后, 我们对状态空间图进行遍历, 生成若干路径作为测试用例. 另一方面, 我们将规约中的各类要素匹配到系统实现中, 并根据匹配结果对实现源码进行插桩, 在相应动作位置加入拦截器 (interceptor). 在受控测试阶段, 我们将插桩后的实现以集群模式进行部署运行, 并使之与测试部件进行交互. 我们一共设计了 5 个测试部件即测试驱动、动作调度部件、状态监控部件、故障注入部件及客户端. 测试驱动负责控制整个测试流程, 动作调度部件负责控制各动作的执行顺序, 状态监控部件能够监控并检查被测系统的运行状态, 故障注入部件被用来向被测系统中的特定程序位置注入各种类型的故障, 客户端用来在特定程序位置发起各类用户请求.

我们使用图 2 中的示例来解释测试部件使用测试用例与被测系统进行交互的过程. 首先, 测试驱动负责读入测试用例并部署并运行被测系统集群. 测试用例是一条由状态和动作组成的路径. 在该路径中, 边代表不同的动作, 点代表动作执行后的预期状态. 在被测系统运行时, 由动作调度部件控制集群中的动作发生顺序. 例如, 第 1 个动作 *RequestVote*(*N*1, *N*2) (本文使用首字母大写英文表示 TLA+规约中的动作及变量, 使用首字母小写英文表示对应源码中的对象) 表示节点 1 向节点 2 发送了一条 *requestVote* 消息. 相应地, 动作调度部件在检测到相应的动作被触发时, 会优先将其放行以执行该动作. 状态监控部件会在动作执行完毕后收集被测系统集群中的状态值, 并将其与测试用例中的预期状态进行比较. 当测试用例显示下一个动作是向被测系统集群中注入故障时, 故障注入部

件会在指定节点上模拟注入相应类型的故障如重启. 当测试用例中的下一个动作是发起用户请求时, 客户端模块会使用相应的请求参数模拟用户请求.

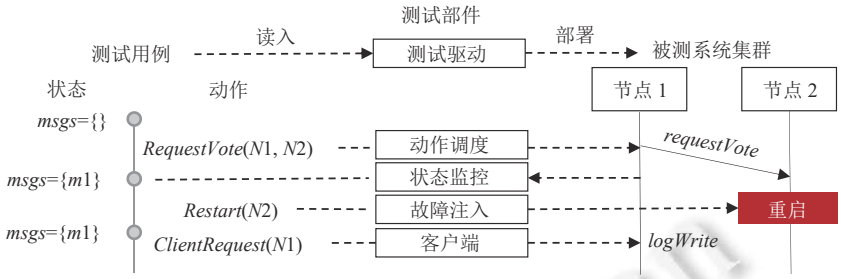


图 2 一个简化的测试 Raft 实现的示例

2.2 映射 TLA+规约到系统实现

对于反映了被测系统设计的 TLA+形式化规约, 我们将其中的变量、动作和常量映射到相应的源码实现中. 由于这些要素与程序语义高度相关, 我们很难自动化地进行映射. 因此, 我们手动地在实现源码中标注出变量和动作, 使得测试部件能够半自动地在测试过程中对其进行识别和处理.

TLA+规约中包含有多种变量、动作和常量, 我们为此设计了不同的映射方法. 如表 1 所示, TLA+变量包括状态相关变量、消息相关变量、测试计数变量以及辅助变量, 对于前 3 种变量, 可以分别映射为类成员变量、动作调度模块中的集合变量以及测试参数, 而对于辅助变量不做处理. TLA+动作包括一般行为、用户请求以及故障这 3 种类别. 一般行为可映射为系统实现中的对应方法. 用户请求和故障采用系统脚本调用等方式进行映射. TLA+常量中的特殊值常量及状态空间约束常量分别被映射为一个 Map 数据结构和测试超参. 下面我们分别对这些映射方法进行具体说明.

表 1 将 TLA+规约中不同类别要素映射到对应系统实现的方法

TLA+要素	类别	对应系统实现
变量	状态相关变量	类成员变量
	消息相关变量	动作调度模块中的集合变量
	测试计数变量	测试参数
	辅助变量	—
动作	一般行为	对应方法
	用户请求	系统脚本调用
	故障	代码重写/一般行为重写/系统脚本调用
常量	特殊值常量	Map数据结构
	状态空间约束常量	测试超参

2.2.1 映射变量

基于变量在 TLA+规约中的用途, 我们将所有变量分为了 4 种类型: 状态相关变量, 消息相关变量, 计数变量以及辅助变量. 其中, 状态相关变量描述了被测系统的具体状态, 如 Raft 节点的角色. 消息相关变量可以模拟被测系统节点间的消息通信过程. 测试计数变量被用来为被测系统中的特殊动作即故障和用户请求进行计数. 辅助变量被用作对 TLA+规约进行证明或验证. 我们为这些变量分别设计了映射策略.

状态相关变量: 状态相关变量能够直接被映射到系统实现的对应代码中, 且往往以类成员变量 (class field) 的形式存在. 例如在图 3(a) 的 TLA+规约中, 变量 *State* 表示 Raft 节点的角色 (行 1). 如图 3(b) 行 3 所示, 对于 Raft 实现 Raft-Java[10], 我们将该变量匹配到其 RaftNode 类中的 *state* 字段 (行 3). 我们需要为该字段添加标注*@Variable*, 并使用变量名 *State* 作为标注值 (行 2).

```
1. VARIABLE State
2. BecomeLeader(i) ==
3.    /\state' = [state EXCEPT ![i] = Leader]
```

(a) Raft 规约中的 *State* 变量及 *BecomeLeader* 动作

```
1. public classRaftNode{
2. +   @Variable("State")
3.     private NodeStatestate=NodeState.STATE_FOLLOWER;
4. +   public static NodeStateVAR$state
5. +       = NodeState.STATE_FOLLOWER;
6.
7.     private void becomeLeader() {
8.         …
9.         state = NodeState.STATE_LEADER;
10. +      VAR$state= NodeState.STATE_LEADER;
11.        …
12.    }
13. }
```

(b) Raft-Java 中的对应源码及插桩

图 3　为变量 *State* 进行匹配及插桩

在测试运行时, 测试驱动会自动地会每一个标注过的状态相关变量增加一个影子变量 (行 4). 当状态相关变量的值被初始化 (行 3) 或更改时 (行 10), 影子变量的值也会被赋予相同的值 (行 5 与行 11). 通过这样的方式, 状态监控模块在读取变量值时, 可以直接读取影子变量的值而不会影响到原有状态相关变量的读写. 如图 3(a) 中变量 *state* 的值在动作 *BecomeLeader* 中被更改为 *Leader*. 该变量的实现在图 3(b) 中第 3 行被初始化为 *STATE_FOLLOWER*, 在第 9 行被更改为 *STATE_LEADER*. 相应地, 测试驱动添加了影子变量 *VAR$state* (行 4) 并将其初始化为 *STATE_FOLLOWER* (行 5). 并在变量 *state* 值更改时置为相同的值 *STATE_LEADER* (行 10).

消息相关变量: 消息相关变量不存在相应的实现代码. 其在 TLA+规约中被用来模拟节点间的消息通讯流程, 存储当前系统中所有未处理 (on-the-fly) 的消息. Raft 规约中使用集合 *msgs* 作为消息相关变量, 其中存储的消息元素包含消息的具体内容、消息发送节点及消息接收节点. 向 *msgs* 中加入一个消息元素时, 表示消息发送节点发出了一条消息. 当该消息元素从 *msgs* 被移除时, 表示消息接收节点收到了该消息.

我们在动作调度中模块实现了一个集合变量作为消息相关变量对应的系统实现. 当某节点向其他节点发送消息时, 其消息发送动作会将消息元素的相关信息发送至该集合中. 当某节点接收到一条消息时, 消息接收动作会将消息元素发送至动作调度模块, 将对应的消息元素从集合中移除. 更多消息相关动作的细节在第 2.2.2 节中介绍.

测试计数变量: 测试计数变量在系统实现中同样不存在相应的代码. 其在 TLA+规约中被用来对用户请求和故障类型的动作进行计数. 并通过限制动作数量以达到限定状态空间大小的效果. 我们将测试计数变量映射为测试过程中的运行时测试参数, 用来计算向被测系统中注入的故障数量和发起的用户请求数量.

辅助变量: TLA+形式化规约除了可用于模型检查外, 还可用于性质证明. Raft 的 TLA+规约中同样有一些变量专用于性质证明. 例如, 变量 *allLogs* 存储了向集群中所写入的所有历史数据, 可以用于证明日志单调性的证明过程中. 但该变量不存在于真实的状态空间中, 即在真实系统里没有任何对应的实现. 因此, 我们在测试中直接忽略这些用于证明的辅助变量.

2.2.2　映射动作

根据 TLA+规约中的动作在真实系统内的行为种类, 我们将所有的行为分为 3 类: 一般行为、用户请求及故障, 并为映射这 3 类行为到实际系统中设计了不同的策略.

一般行为: 一般行为能够直接映射到系统实现中的方法. 如图 4(a) 所示, TLA+规约中的动作 *RequestVote*(*i*, *j*) 定义了节点 *i* 发送给节点 *j* 一个投票请求消息. 其中具体的消息发送行为及消息的内容在行 3–8 中定义. 消息内容包括消息类型 *mtype*, 选举任期号 *mterm*, 最新日志所属任期号 *mlastLogTerm* 及索引号 *mlastLogIndex*, 以及消息发送节点 *i* 和接收节点 *j*. 我们在 Raft 实现 Raft-Java 中找到对应的方法 *requestVote*, 如图 4(b) 所示. 我们对该方法使用标注*@Action*, 并使用动作名 *RequestVote* 作为标注值 (行 1). 此外, 由于动作 *RequestVote* 拥有两个参数 *i* 与 *j*, 我们也需要在代码实现中手动为其收集两个参数的实时值. 因此, 我们使用接口 *Action.getParams* 将两个值放置在对应的参数位置 (行 8). 此外, 我们还需要为消息相关的行为收集消息内容. 因此, 我们在参数收集接口中额外写入代码 (行 9–14) 以收集消息内容的实时值. 需要注意地是, 消息内容的排列顺序应与 TLA+规约中的消息内容排

列顺序一致, 即{消息类型, 选举任期号, 最新日志所属任期号, 最新日志索引号, 消息发送节点, 消息接收节点}, 以方便动作调度模块对匹配的动作自动进行识别和比对.

```
1. RequestVote(i, j) ==
2.   /\state[i] = Candidate
3.   /\Send([mtype|→ RequestVoteRequest,
4.           mterm|→ currentTerm[i],
5.           mlastLogTerm|→ LastTerm(log[i]),
6.           mlastLogIndex |→ Len(log[i]),
7.           msource      |→ i,
8.           mdest        |→ j])
```

(a) Raft 规约中的 *RequestVote* 动作

```
1. +@Action("RequestVote")
2.  private void requestVote(Peer peer) {
3.    …
4.    requestBuilder.setServerId(localServer.getServerId())
5.            .setTerm(currentTerm)
6.            .setLastLogIndex(getLastLogIndex())
7.            .setLastLogTerm(getLastLogTerm());
8. +  Action.getParams(this.NodeId, peer.NodeId,
9. +          "RequestVoteRequest",
10. +         currentTerm,
11. +         getLastLogTerm(),
12. +         raftLog.getLastLogIndex(),
13. +         this.NodeId,
14. +         peer.NodeId);
14. +  ActionScheduler.notifyActionAndBlock();
15.   …
16. +  StateMonitor.checkVariables();
17.  }
```

(b) Raft-Java 中的对应源码及插桩

图 4 为动作 *RequestVote* 进行匹配及插桩

在测试运行时, 测试驱动会自动地在每一个动作中添加两行代码, 即 *ActionScheduler.notifyActionAndBlock* 与 *StateMonitor.checkVariables*. 前者在参数收集语句 *Action.getParams* 后插入, 会将当前动作的名称、收集到的参数发送给动作调度模块, 并将当前的线程挂起以等待动作调度部件返回的调度通知. 后者在方法末尾插入, 会收集当前在该节点上所有影子变量的实时值, 并发送给状态监控部件进行状态检查.

用户请求: 用户请求在 TLA+规约中与其他动作类型一样, 同样以方法的形式定义. 但在系统实现中属于由用户发起的外部行为, 并没有直接对应的代码实现. 用户通常通过使用系统所提供的脚本文件, 并输入不同的参数来使用系统所提供的各种服务, 例如向 Raft 集群中写入键值对. 图 1 中的客户端部件将这些用户请求脚本进行集成. 当动作调度模块发现测试用例中出现用户请求动作时, 则由客户端部件使用用例中的参数调用相应脚本以模拟发起请求.

例如, Raft 的 TLA+规约中定义了动作 *ClientRequest*(*i*), 表示向领导者节点 *i* 中写入一条日志. 相应地, 在 Raft-Java 系统中, 我们可以使用脚本 run_client.sh 来向集群中写入一个 *Key-Value* 键值对:

./run_client.sh $*Cluster* $*Key* $*Value*

注意 *ClientRequest* 动作中并没有定义写入日志中具体的数据为何值. 在行为内部逻辑中, 规约开发者使用测试计数变量 *clientRequests* 来区别不同的用户请求. 每当该行为被执行一次, 则将 *clientRequests* 的值加 1. 而在测试中使用脚本发起用户请求时, 我们需要具体的 *Key* 和 *Value* 值来写入集群中. 我们通过将抽象的测试计数与具体的写入数据值进行匹配, 来对用户请求动作使用的参数进行映射. 例如, 我们使用 *clientRequests* 变量在状态空间中不同的值作为客户端部件发起的 *ClientRequest* 动作的参数, 在第 1 个用户请求动作中, 写入键值对 (1, 1), 在第 2 个动作中写入 (2, 2).

故障: Raft 的 TLA+规范中显式地定义了 4 种类型的故障, 即节点通信超时、消息丢失、消息重复、节点重启. 我们模拟了这 4 种故障行为, 并使用故障注入部件在测试过程中主动向被测系统中注入特定的故障.

节点通信超时故障在 Raft 协议中能够发起新一轮的选举. 系统实现通常在会使用计时器来检测该类型故障的发生. 当节点在计时器结束前没有收到来自主节点的消息时, 则视为该节点发生了节点通信超时故障, 并进入主节点选举流程. 然而, 由于各节点上的计时器被设定为随机值, 因此在系统初始化时我们无法预知节点通信超时故障会首先发生在哪一节点上. 因此, 为了使测试过程遵循测试用例中规定的动作序列, 我们需要对节点通信超时故障的不确定性进行控制. 具体地, 我们将计时器进行屏蔽, 并通过动作调度部件直接选择特定节点发起新的选举轮次.

对于消息相关故障, 我们对消息处理函数进行插桩以进行故障模拟. 针对消息丢失故障, 我们简单地忽略掉函数体并直接返回. 针对消息重复故障, 我们使用相同的函数参数重复执行两次函数体. 由于不同的系统使用不同的消息通信机制, 因此我们手动地选择需要插桩的消息处理函数. 例如, Raft-Java 使用基于 RPC 的通信, 因此我们可以直接插桩 RPC 代理函数. Xraft 使用基于 UDP 的通信, 因此我们选择插桩 UDP 包接收的函数.

对于节点重启故障, 我们通过一个可以杀死节点进程并使用相同配置启动新节点进程的脚本进行模拟. 当测试用例中发生一个节点重启故障动作时, 我们将节点相关参数即进程 ID 和节点配置输入到故障注入部件中, 并使用这些参数调用该脚本进行故障注入.

2.2.3 映射常量

开发者在 TLA+规约中使用常量来达到两个目的: 定义特殊的变量值或动作类型, 以及在模型检查中约束状态空间大小. 我们为这两类常量设计了不同的映射策略.

特殊值常量: 这些常量定义了规约中的特殊值. 例如, 常量 *Follower*、*Candidate*、*Leader* 是变量 *state* 可取的 3 个值, 定义了 Raft 节点的角色. 常量 *RequestVoteRequest*、*RequestVoteResponse*、*AppendEntriesRequest*、*AppendEntriesResponse* 定义了 Raft 中不同的消息类型, 用来区分不同的消息相关动作. 对于特殊值常量, 我们在测试驱动部件中使用一个 Map 数据结构来存储其在 TLA+规约中以及在代码实现中的值. 在测试过程中, 如果动作调度部件需要分辨不同的动作类型, 或者状态监控部件需要判断规约与实现中的变量值是否匹配, 则可以直接访问该 Map 进行常量值查询.

状态空间约束常量: 分布式系统理论上由于存在无穷的节点数量, 可能发生无穷的用户请求及不确定的故障, 因此可以有着无尽的状态空间. 因此, 开发者会使用状态空间约束常量来对这些因素进行限制. 当对 TLA+规约进行模型检查时, 开发者对这些常量进行赋值, 以使模型检查过程能够在探索完特定场景后终止. 例如, 在 Raft 的 TLA+规约中, 我们可以使用{*n*1, *n*2, *n*3}作为常量 *Server* 的值, 表示我们在进行模型检查的场景中构建了一个拥有 3 个节点的集群. 常量 *ClientRequestLimit* 定义了在模型检查过程中能执行的 *ClientRequest* 动作的最大次数. 我们将该常量与测试计数变量 *clientRequests* 一起使用, 并限定 *clientRequests* 的值不得大于该常量值. 除此以外, 我们还在规约中加入了 *TimeoutLimit*、*RestartLimit*、*DuplicateMessageLimit*、*DropMessageLimit* 这 4 个状态空间约束常量以及其相应的测试计数变量, 用来限制故障动作的发生次数. 对于所有的状态空间约束常量, 我们将其作为测试超参使用, 并将这些值存储于测试驱动部件中.

2.3 测试用例生成

如第 1.3 节介绍, 通过使用 TLC 对分布式系统的 TLA+规约进行模型检查, 可以得到一张状态空间图. 我们对该图进行遍历以生成测试用例. 每一个测试用例是从初始状态出发的一条有向路径. 需要注意的是, 通过遍历状态空间图生成测试用例有多种策略, 如节点覆盖导向、边覆盖导向以及路径覆盖导向. 节点覆盖导向策略旨在覆盖状态图中全部的状态点, 能够生成最少的测试用例数量, 但是会漏掉许多系统行为即路径. 路径覆盖导向策略旨在覆盖状态图中存在的全部系统行为, 是完备的策略用例生成方法. 但由于状态图中环的存在, 会生成无限长的测试路径和无穷多的测试用例. 例如, 如果存在两个不同的动作 *Crash*(*N*) 与 *Reboot*(*N*) 分别表示对于节点 *N* 的宕机和重启, 且两个动作在状态空间图中构成一个环, 则在一个测试用例中, 可以通过无限地对环进行遍历即无限宕机和重启同一个节点以生成无限长的路径. 因此, 我们采用了边覆盖导向的测试用例生成策略, 可以在生成有限测试用例的同时测试尽可能多的系统行为.

我们使用一个简单的深度优先算法来实现对状态空间图的边覆盖导向遍历. 如算法 1 所示, 我们从初始状态节点出发开始遍历 (行 3), 并对其后继节点进行迭代遍历 (行 4–15). 遍历过程在两种情况下会终止: 当前状态节点的所有后继边都已经被访问过, 或者当前状态节点为终止状态. 终止状态的定义在不同的测试场景中不同. 例如, 在节点选举场景中, 当一个状态由动作 *BecomeLeader* 生成, 代表该节点被选举为领导者节点, 则该状态即为终止状态. 而在日志复制场景中, 终止状态是由 *AdvanceCommitIndex* 动作生成的状态, 其代表着领导者已经提交了一条日志. 若上述两个终止条件任一被满足, 则将当前路径 *path* 加入到测试用例集中 (行 6), 否则继续进行遍历. 对

于当前节点的每一条出边, 若其在之前被访问过, 则直接跳过该边 (行 10, 11), 否则将其标注为 *visited* (行 13), 将该边以及相应的后继节点加入到路径中 (行 14), 并继续基于后继节点进行遍历 (行 15).

算法 1. 边覆盖导向的测试路径生成.

输入: 状态空间图 *g* 及初始节点 *initState*;
输出: 路径集合 *testCases*.

```
1. testCases← ∅
2. initPath ← new Path(initState)
3. traverse(initState, initPath, g)
4. Function traverse(state, path, graph) do
5.     if isEndState(state) ∨ allOutEdgeVisited(state) then
6.         testCases.add(path)
7.         return
8.     foreach succ∈state.successors do
9.         edge ← graph.edge(state, succ)
10.        if edge.visited = TRUE then
11.            continue
12.        else
13.            edge.visited ← TRUE
14.            path.add(edge, succ)
15.            traverse(succ, path, graph)
```

通过对状态空间图的遍历, 我们获得一个测试用例集合. 集合中的每一个用例都是一条路径. 路径中的边表示发生在测试过程中的动作. 节点代表在系统实现中进行状态检查的程序点.

2.4 受控测试

在测试阶段, 我们为算法 1 中生成的每一个测试用例发起一轮测试并进行初始化. 具体地, 测试驱动部件首先读入该用例并获取测试相关信息, 即动作序列、动作执行后期望的状态、需要发起的用户请求类型及参数、需要注入的故障类型及参数. 随后, 测试驱动使用超参所规定的节点数量运行并初始化被测系统集群. 在此过程中, 测试驱动会根据开发者所做的变量及行为映射进行自动插桩, 并将测试计数相关变量所映射的测试参数初始化.

在受控测试过程中, 测试框架会根据在动作中插桩的代码对动作执行顺序进行控制并监控动作执行后的状态. 如图 5 所示, 当被测系统运行到被映射的动作 *ActionA* 的相关代码时, 即触发插桩的 *ActionScheduler.notifyActionAndBlock* 语句. 该语句会将当前线程挂起, 并将动作的相关信息发送至动作调度 (*Action Scheduler*) 部件. 这里的两个节点 1 和 2 上的代码都需要执行 *ActionA*, 并发送了相应的消息到动作调度部件. 图 5 中测试用例当前需要执行的动作是 *ActionA*(*N*1), 即首先执行节点 1 的 *ActionA* 动作. 因此, 动作调度部件回复 (*Reply*) 节点 1, 并恢复挂起的进程继续执行. 当动作执行完毕时, 会触发插桩的 *StateMonitor.checkVariables* 语句. 该语句会收集当前节点上的所有变量值并发送至状态监控部件进行检查. 当状态值符合测试用例中的状态变化时, 则继续控制执行测试用例中的下一动作.

在整个测试过程中, 动作调度部件和状态监控部件会根据被测系统的行为判断其系统实现是否与 TLA+规约一致. 如果一个动作不存在于测试用例的动作序列中, 或者本该被执行的动作从未在被测系统中发生, 则动作调度部件会将其报告为一个系统实现与形式化规约之间的不一致异常. 另一方面, 如果任何变量值与测试用例中的状态变量值不同, 则状态监控部件会将其报告为一个系统实现与形式化规约之间的不一致异常. 需要注意的是, 节点

重启故障并没有相应的状态检查程序点. 其预期状态在下一行为中被检查. 当一个不一致异常被报告时, 测试驱动会为开发者生成一份测试报告, 其中包含了整个测试用例的信息以及具体的不一致的动作或变量信息.

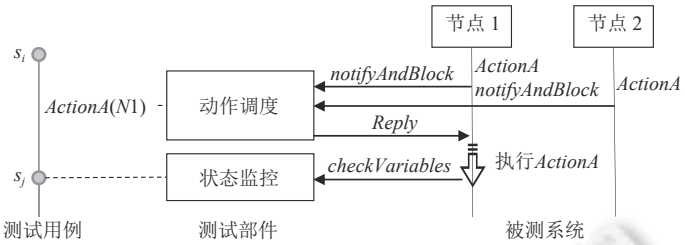


图 5 受控测试流程

针对报告的不一致异常, 我们需要根据测试报告对其进行进一步的诊断, 以识别该不一致是由系统实现中的缺陷造成的或是 TLA+规约中的缺陷造成的. 我们通过手动检查测试报告, 并将测试用例中的路径与实现代码相比较, 以判断是 TLA+规约或是系统实现违背了系统设计. 对于无法判断的不一致异常, 我们会直接请求系统开发者的帮助以讨论并弄清其设计意图.

除此以外, 测试人员的人工错误也会导致报告错误的不一致异常, 即误报 (false positive). 我们在实践过程中共总结了两类人工错误. 首先, 测试人员在使用标注映射状态相关变量或一般行为时, 可能会将标注中的变量名称、动作名称、动作参数误写, 从而导致报告不存在于测试用例中的动作或不一致的状态. 其次, 测试人员在进行映射时, 可能会构建错误的映射关系, 如对某个动作没有标注, 则在测试中动作调度部件会报告某个动作未发生. 人工错误只能在系统测试的过程中被发现. 因此, 我们需要进行多轮测试以消除人工错误. 当我们在一轮测试中发现不一致异常时, 需要对异常的根因进行诊断以判断引发异常的是被测系统实现缺陷、TLA+规约缺陷或人工错误. 如果该异常由人工错误引发, 我们修复该错误, 重新生成测试用例并进行下一轮的测试. 此类人工错误通常很容易修复, 但是多轮测试会浪费许多测试资源. 在未来的工作中, 我们会设计更好的策略以避免人工错误并降低测试成本.

2.5 方法实现

我们将整个测试方法以大约 3 000 行代码的代价进行了实现, 包括标注框架、运行时插桩、测试用例生成以及图 1 所示的 5 个测试部件. 被用来映射 TLA+变量与动作的部分基于 java.lang.annotation 包实现. 运行时插桩由 Java 字节码操作和分析框架 ASM[27]提供支持. 测试用例生成部分则使用了 Python 的 NetworkX[28]包来对状态空间图进行遍历. 除此以外, 我们还利用大约 300 行代码来对 Raft 中特有的变量、动作及常量进行维护, 并在 Xraft 及 Raft-Java 中添加了 52 行代码来标注其变量和动作.

3 实验评估

为了评估基于 TLA+形式化规约的分布式系统测试方法的可行性和有效性, 我们在两个开源的基于 Raft 实现的分布式系统上进行了实验.

3.1 实验设置

被测系统: 我们从 Raft 官方网站[29]上选取了两个分布式系统 Xraft[11]及 Raft-Java[10]进行了实验. 首先, 我们挑选出使用 Java 语言构建的 Raft 项目 (30 个). 随后, 我们阅读项目的 README 文件, 识别并剔除其中 Raft 协议不能作为独立部件运行的项目 (15 个). 例如, 在 Hazelcast[30]平台中, Raft 协议被内嵌在其节点实现中. 对于此类项目, 我们无法通过一个简单的客户端去驱动 Raft 中的 4 个场景. 因此, 我们不能直接测试这些项目中的 Raft 实现. 进一步地, 我们逐一阅读剩余的 15 个项目的代码, 并保留从实现上大致遵照 Raft 规约的项目 (10 个). 对于在原始 Raft 协议基础上做了极多修改的项目如 SOFAJRaft[31], 我们将其从候选对象中剔除. 最后, 我们筛选掉已经归档或多年不再活跃的项目. 最终剩余 Xraft 与 Raft-Java 两个项目. 其中, Raft-Java 是 GitHub 上获取 Star 数量超过一千的流行项目. Xraft 流行度稍低 (186 Stars), 但该项目在一本已出版的 Raft 书籍[32]里被用作示范项目, 因此具有一

定的代表性. 我们使用了 Xraft 的最新版本实现, 以及 Raft-Java 存在已知缺陷的两个旧的版本.

规约修改: 在本文中, 我们主要使用 Raft 作者所开发的 TLA+规约. 该规约中仅对领导者选举及日志复制场景进行了建模. 我们额外实现了日志精简场景相关的 TLA+代码. 由于成员变更场景在 TLA+中较难实现, 目前我们没有针对该场景进行测试.

此外, 我们所挑选的两个 Raft 实现在某些设计细节上与 TLA+规约有一些不一致. 例如, Raft-Java 在实现中采用了同步通信机制, 即当一个节点发送消息后, 在收到相应的回复消息并处理之前, 其不能运行其他的消息相关动作. 因此, 我们在针对 Raft-Java 的 TLA+规约中将消息发送动作和其对应的消息接收动作合并为了单个动作. 针对由 TLA+规约中的缺陷导致的不一致异常报告, 我们将缺陷修复后再重新运行整个测试流程.

实验设施: 我们在一台配备了 3.1 GHz 的 Intel Core i9 CPU、64 GB 内存的 Windows 工作站上进行了实验. 其中, 为 TLC 模型检查器分配了 32 GB 的内存. 测试过程在两个运行在 VMware Workstation Pro 16.1 平台内的 Linux 虚拟机实例中进行, 每一个实例分配了两个 CPU 核心和 8 GB 的内存. 所有的 Raft 系统都以伪集群的方式进行部署, 即每一节点都以进程的方式运行在同一台主机上. 对于所有的测试场景, 我们设置了以下的测试超参: 每个集群中有 5 个节点, 在单轮测试中出现的所有故障不超过 5 次, 写入日志的用户请求最多执行 3 次.

3.2 实验结果

在实验中我们一共发现了 7 个缺陷, 其中 5 个是 Raft 实现中的缺陷, 2 个是 Raft 的 TLA+规约中存在的缺陷. 在 Raft 实现中的 5 个缺陷中, 3 个是 Xraft 中的未知缺陷, 2 个是 Raft-Java 中的已知缺陷. 接下来我们对每一个缺陷进行详细说明.

未知缺陷: 我们在 Xraft 的主节点选举场景中发现的第 1 个缺陷[33]如图 6 所示. 其根本原因是开发者在实现中对变量 *votesGranted* 进行了过度的简化. 该变量存在于候选者节点上, 被用于记录已经承认其选举资格的节点信息. 然而, Xraft 简单地将该变量实现为整数 *votesCount*: 每当候选者节点收到一个来自其他节点的选票时, *votesCount* 的值即被加一. 在这个缺陷场景里, 节点 2 首先成为候选者并将 *votesCount* 的值置为 1. 随后, 节点 2 发送 *RequestVote* 请求至节点 1 且成功地收到了选票. *votesCount* 的值被置为 2. 然而, 一个消息重复故障使得节点 2 收到了两条来自节点 1 的选票消息, 导致其 *votesCount* 的值增加为 3. 由于集群节点数量为 5, 节点 2 认为此时已经收到了大部分节点的回复, 因此错误地成为了领导者.

图 7 展示了在 Xraft 领导者选举场景中发现的另一个缺陷[34]. 在该缺陷场景中, 节点 1 和节点 4 同时成为候选者节点. 节点 1 首先向节点 2 和节点 3 发送了投票请求, 而来自节点 4 的投票请求消息被延迟了. 节点 2 认可了节点 1 的投票请求, 将变量 *votedFor* 置为 *N*1 以记录其已经为节点 1 投票. 随后, 节点 2 发生重启故障, 并错误地将 *votedFor* 变量的值重置. 在 Raft 的 TLA+规约中, 该变量应该在重启前被持久化, 使得在节点重启后仍然能够读到持久化的值. 由于节点 2 此时丢失了其投票记录, 当其收到来自节点 4 迟到的投票请求信息时, 节点 2 将其选票投给了节点 4. 而当节点 1 最终收到节点 3 的选票后, 由于认为节点 2 仍然投票给自己, 最终错误地成为领导者.

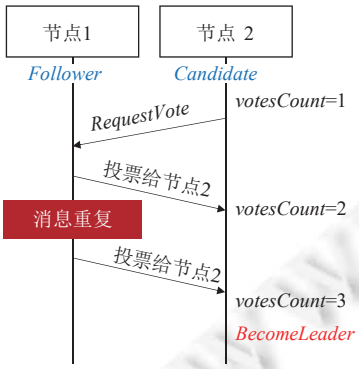


图 6 Xraft 中的未知缺陷导致节点 2 可以在没有大多数节点支持时成为领导者

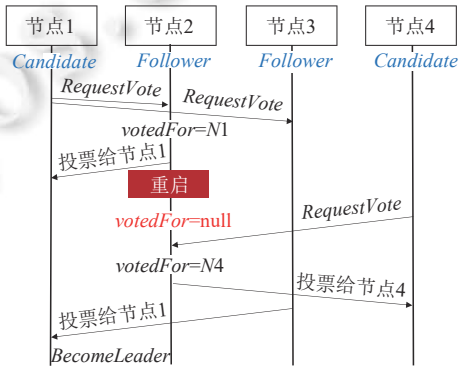


图 7 Xraft 中的未知缺陷导致节点 1 在节点 2 更改选票后仍然可以成为领导者

图 8 展示了 Xraft 中的一个复杂缺陷[35]. 在该缺陷场景中, 节点 1、2、3 在最开始同时成为候选者, 且任期 *term* 相同皆为 1. 节点 1 在收到节点 4、5 的投票回复后成为领导者节点, 并通过 *AppendEntry* 消息通知其他的节点. 接收到该消息的节点会写入一条没有任何数据的日志即 *NoOp* 日志, 以表示该节点已经为其他节点投过选票. 在将 *NoOp* 日志持久化后, 节点 2 和节点 3 从候选者成为了跟随者, 并将其投票信息 *votedFor* 置为空. 而节点 4 发生了重启故障并成为候选者, 节点上的变量 *term* 重新置为 1, *votedFor* 记录为投给自己 *N*4, 写入的 *NoOp* 日志由于持久化仍然存在. 接着, 节点 4 将自己的信息写入 *RequestVote* 消息中并请求节点 2 和 3 的投票. 在 Xraft 中, 跟随者会当同时满足 3 个条件时更改自己的投票: 第一, 当前任期值小于等于投票请求的任期值; 第二, 当前的 *votedFor* 变量值为空; 第三, 节点上存储的数据日志并不比投票请求中的数据日志版本更新. 节点 2 和节点 3 不应该认可节点 4 的投票请求, 因为其传递的日志信息为 *NoOp* 日志而非数据日志. 但错误的代码实现使得节点 4 收到了两个节点的投票, 并在节点 1 仍为领导者的情况下成为集群中的第 2 个领导者.

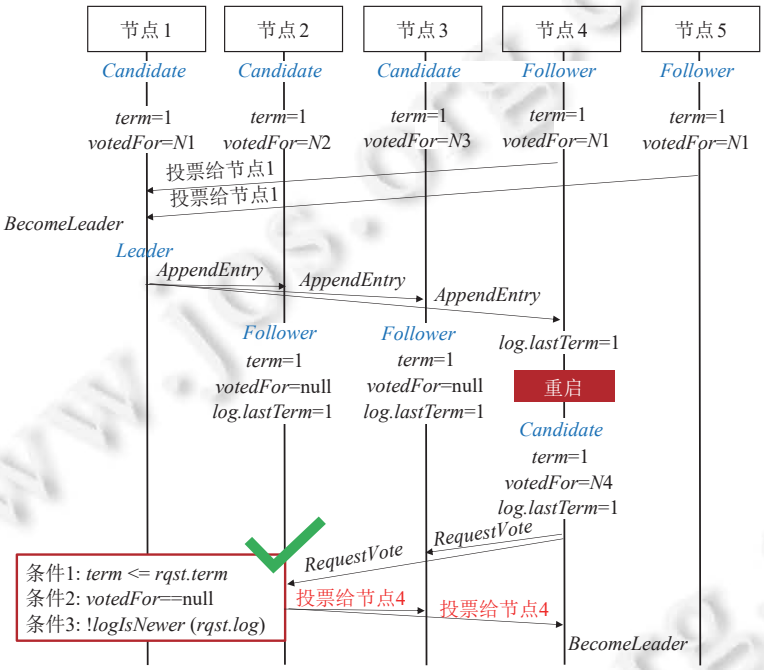


图 8 Xraft 中的未知缺陷导致节点 1 和节点 4 在同一任期成为领导者

已知缺陷: 由于我们没有在 Raft-Java 中发现任何未知缺陷, 为了评估方法对于 Raft-Java 是否有效, 我们在其项目页面中浏览了其所有的 36 个历史问题 (issue), 通过人工对问题中的描述及修复代码进行检阅, 分析问题的根本原因, 最终发现了两个违背了 Raft 设计的已修复缺陷. 我们基于该项目的最新版本将修复代码进行回滚, 并针对两个带有缺陷的实现版本进行了实验. 最终我们成功地复现了这两个缺陷. 下面我们对这两个缺陷进行详细说明.

图 9 中的缺陷[36]发生在 Raft-Java 的领导者选举中. 在该缺陷里, 候选者节点发送了一条 *RequestVote* 消息到一个拥有更大 *term* 值的领导者节点. 领导者没有认可该投票请求, 并回复了自己的 *term* 值. 候选者收到该回复后的处理逻辑如图 9 所示, 首先检查自己发送的请求 *request* 里的 *term* 值是否已经过期, 即比自己现有的 *term* 值小 (行 2). 若已经过期则不用处理该请求对应的回复 *response* (行 3). 如果请求没有过期, 则继续检查

```
1. -if(myTerm  != response. term) {
2. + if(myTerm  != request. term) {
3.     return;
4.  }
5.  if(myTerm< response. term) {
6.     becomeFollower();
7.  }
```

图 9 Raft-Java 中的已知缺陷代码及修复方式

response 中的 *term* 值 (行 5), 如果该值比自己现有的 *term* 值大, 则说明集群内存在更高等级的领导者, 该节点转变为跟随者 (行 6). 然而, 开发者错误地将第 1 次 *term* 检查时的 *request* 误写为 *response* (行 1), 最终导致候选者节点不能成功将角色转变为跟随者并对外提供服务.

另一个 Raft-Java 的缺陷[37]发生在日志精简场景中. 当在 Raft-Java 中生成快照时, 节点会将两个关键的日志元数据相关变量 *lastIncludedIndex* 和 *commitIndex* 进行持久化. 当节点发生重启故障且丢失其状态机中的数据时, 会从持久化的快照中读取数据. 索引值小于 *lastIncludeIndex* 的日志数据会被直接丢弃, 大于 *lastIncludeIndex* 小于 *commitIndex* 的日志数据会被应用于状态机中. 然而, 开发者在重启时错误地将 *lastIncludeIndex* 的值赋给了 *commitIndex* 变量, 导致重启时不会有任何的数据被恢复.

TLA+规约缺陷: 在经过所有作者的诊断和讨论后, 我们确认了两个报告的不一致异常为 TLA+规约中的缺陷. 这些缺陷不会影响模型检查的过程, 但是生成的测试用例中包含了系统实现中不能进入的状态.

图 10 展示了第 1 个规约缺陷. *UpdateTerm*、*HandleRequestVoteRequest*、*HandleRequestVoteResponse* 由析取操作符连接, 表示 *UpdateTerm* 能作为一个单独的动作与 *HandleRequestVoteRequest* 和 *HandleRequestVoteResponse* 进行交错. 但在系统实现中, 变量 *term* 只能在后两个动作内部进行更新. 因此该缺陷导致在测试中不会出现一个单独的 *UpdateTerm* 动作. 我们通过将 *UpdateTerm* 移到后两个动作内部修复了该规约缺陷.

第 2 个规约缺陷出现在动作 *HandleAppendEntriesRequest* 中. 如图 11 所示, 该动作在处理请求时因不同情况有不同的分支, 即拒绝请求 (行 2)、更改节点角色至跟随者 (行 3, 4)、接受请求 (行 5–7). 然而, 只有在第 1 和第 3 个分支调用了 *Reply* 方法. 这导致当 *HandleAppendEntriesRequest* 执行其第 2 个分支时, 其不会直接回复该消息, 而是需要执行第 2 遍 *HandleAppendEntriesRequest* 在第 3 个分支中调用 *Reply*. 在系统实现中, 这些步骤在单个动作中就全部执行完毕了. 为了修复该规约缺陷, 我们在第 2 个分支中加入了一个 *Reply* 方法.

```
1. Receive(m) ==
2.   LET i == m.mdest
3.       j == m.msource
4.   IN  \/ UpdateTerm(i, j, m)
5.       \/ /\ m.mtype = RequestVoteRequest
6.          /\ HandleRequestVoteRequest(i, j, m)
7.       \/ /\ m.mtype = RequestVoteResponse
8.          /\ HandleRequestVoteResponse(i, j, m)
```

图 10 错误的动作交错关系导致了 Raft 中的规约缺陷

```
1. HandleAppendEntriesRequest(i, j, m) ==
2.   \/ Reply(…) \* reject request
3.   \/ /\ \* return to follower state
4.      /\ state' = [state EXCEPT ![i] = Follower]
5.      \/ /\ \* accept request when it is a follower
6.         /\ state[i] = Follower
7.         /\ Reply(…)
```

图 11 Reply 方法的调用缺失导致了 Raft 中的规约缺陷

3.3 性能分析

为了衡量方法的性能, 我们在进行实验时记录了一些度量指标, 包括状态的数量和生成的路径数量, 以及对每个测试场景进行模型检查及测试的时间. 表 2 展示了这些指标的统计结果. 由于 Xraft 采用了异步通信机制而 Raft-Java 使用同步通信, 使得 Xraft 在相同被测场景下比 Raft-Java 更为复杂. 随着被测系统及被测场景越来越复杂, 其产生的状态空间也随之更大 (状态数列显示从 158 到 91 532), 且耗费更多的测试时间 (测试列显示从 6 min 21 s 到 48 h+). 由于 Xraft 的日志同步和日志精简场景过于庞大, 我们并没有对这两个场景的所有用例进行测试. 我们仅针对其产生的前 10 万个用例进行了 48 h 的测试.

表 2 状态空间大小及所耗时间统计

被测系统	被测场景	状态空间		时间	
		状态数	路径数	模型检查	测试
Raft-Java	选举	158	74	2 s	6 min 21 s
	选举+同步	671	2 163	13 s	3 h 43 min 50 s
	选举+同步+精简	1 047	5 729	20 s	10 h 47 min 34 s
Xraft	选举	2 791	17 544	37 s	24 h 22 min 11 s
	选举+同步	35 454	100 000+	2 min 3 s	48 h+
	选举+同步+精简	91 532	100 000+	7 min 51 s	48 h+

4 相关讨论

4.1 方法局限性

完备性: 我们的方法并不能发现分布式系统当中的所有缺陷. 首先, TLA+规约是对分布式系统进行的抽象建模, 其只关注系统实现中重要协议的重要变量及动作, 而不能对所有语句的预期行为进行定义和检查. 相应地, 我们的测试方法对于 TLA+规约中所忽略掉的系统行为也不能进行有效测试, 无法判断在这些行为中在测试中是否出现了缺陷. 其次, 在第 2.3 节中我们使用了边覆盖策略来对生成的路径进行削减, 虽然最后生成的测试用例覆盖了其状态空间中的所有动作, 但被削减掉的路径仍可能存在隐藏的缺陷.

通用性: 我们的方法在经过一定的适配后可以被用于测试其他类型的分布式系统. 其适配代价主要存在于将 TLA+规约映射到系统实现这一步. 在测试工具原型中, 我们共使用了大约 300 行代码为两个 Raft 系统进行适配. 在工具之后的版本中, 我们将进一步地减少该适配代价.

人工代价: 使用测试方法需要测试人员手工将 TLA+规约中的要素映射到对应的系统实现中. Raft 协议的 TLA+规约中共包含 12 个动作和 15 个状态相关变量, 为了测试两个 Raft 系统即 Raft-Java 与 Xraft, 我们分别使用了约 50 行标注相关代码将这些动作和状态相关变量映射到具体的系统源码中. 考虑到两个系统自身的源码数量分别为 15 017 行与 16 530 行, 这些标注带来的人工代价是可接受的.

TLA+规约状态与系统源码状态的对应关系: TLA+规约和系统源码中的状态可能会存在一对多的对应关系. TLA+规约是对系统源码程序逻辑的高层抽象, 因此系统源码中某些具体实现所对应的多个状态可能在 TLA+规约中被映射为了相同的状态. 在这种情况下, 测试方法会遗漏相应的状态检查, 进一步可能遗漏其中潜在的缺陷. 如果 TLA+规约能够尽可能地贴近系统源码实现, 则两者之间状态的对应关系也会随之贴近一对一. 但是构建如此细节化的规约是非常困难的, 且会带来很大的测试代价. 因此, 在使用 TLA+规约对系统实现进行测试时, 我们使规约处于一个中等的抽象层次, 在降低测试代价的同时不会遗漏过多的缺陷.

仅适用于基于 Java 的系统: 我们的方法中的标注及插桩框架分别基于 java.lang.annotation 以及 Java 字节码插桩工具 ASM 实现. 为了将该测试方法应用在其他非 Java 语言实现的 Raft 系统中, 我们需要重新实现测试框架的标注及插桩部件. 然而, 测试框架的其他部件如测试用例生成、测试驱动、动作调度、状态监控、故障注入及客户端可以直接重用于测试其他语言的系统实现.

对有效性的威胁: 由于资源限制及修改 TLA+规约的不易, 我们仅在两个 Raft 系统中验证了测试方法的有效性. 因此, 我们的实验结果可能不能反映其他 Raft 实现中的某些情况. 尽管如此, 我们在测试对象挑选过程中尽可能地保持客观性. 此外, Raft-Java 和 Xraft 两个系统也与原有的 Raft 规约有着一定的差异性. 比如, Raft-Java 实现了一个 *PreVote* 阶段来提高系统活性, 而 Xraft 也利用 *NoOp* 日志机制来修复可能的数据覆盖问题. 两个机制在所有的 Raft 实现中都有一定的代表性且被广泛使用.

4.2 经　验

本方法的研究动机是为了填补 TLA+形式化规约与分布式系统实现之间存在的鸿沟. 在针对 Raft 进行的实践中, 我们发现通过改善 TLA+规约以及代码实现能够缩小该鸿沟, 减少将 TLA+规约映射到系统实现中的代价.

开发规约时正确地映射系统实现: 如第 3.2 节所示, 我们在实验中发现了两个 Raft 的规约缺陷. 我们推测这些缺陷的出现是因为 TLA+规约开发者只想将该规约用作形式化验证或定理证明. 这些缺陷虽然会导致出现系统实现中不存在的状态, 但对于验证或证明是可容忍的. 然而, 当我们将 TLA+规约视为对系统设计的建模时, 这些缺陷是不能容忍的. 当开发者为分布式系统开发规约时, 应该为每一个动作考虑两方面的内容: 该动作是否能与其他动作进行交错?该动作的内部逻辑是否会导致不符合系统实现的状态? 当能够保持动作之间的交错关系以及动作内部的逻辑与实现一致时, 才能保证整个 TLA+规约正确地映射了系统实现.

系统实现如何适应基于 TLA+规约的测试? 当对系统实现进行开发时, 开发者从两个方面可以使得系统实现更易于应用我们的测试方法. 第一, 将关键变量及动作更为显式地声明和规范化的命名. 在将规约映射到系统实现

的过程中, 在系统源码中寻找变量与动作对应的实现有时会十分麻烦. 将变量与动作直接以规约中的名字命名, 且将其实现为类成员变量和方法可以有效地帮助减少映射的人工代价. 第二, 开发者可以提供多种的故障接口. 我们在测试中为不同的系统模拟了不同类型的故障. 为了实现特定的故障类型如超时故障的模拟, 我们对源码进行了比较严重的侵入式修改. 因此, 一些易于使用的故障模拟接口也能够帮助介绍测试代价.

4.3 未来方向

将 TLA+规约映射到分布式系统实现中需要一定的人工代价. 受已有的利用形式化语言如 Coq[38]直接对分布式系统在开发阶段进行验证的工作[18,19,39−41]的启发, 我们计划利用本文中将 TLA+规约与代码实现进行映射的思想, 提出面向验证的分布式系统开发方法. 理想情况下, 分布式系统开发者在进行开发时使用简单的标注及规范化的注释, 随后其相应的形式化规约及测试用例能够在少量的人工代价下自动生成.

另一方面, 在第 2.3 节中我们使用的状态空间遍历方法并不完备. 可能会生成一些无意义的路径或丢失一些隐藏有缺陷的路径. 我们计划通过将传统的形式化状态空间削减方法与从测试代码实现产生的实时反馈相结合的方法, 进一步对状态空间遍历方法进行改善.

5 相关工作

分布式系统形式化验证: 通过对分布式系统进行形式化描述, 开发者可以对分布式系统的正确性进行验证. 部分研究工作利用形式化语言 TLA+[12]和 Coq[38]直接对分布式系统进行建模验证[14,16,18,40,42,43]. 然而, 经过形式化验证的分布式系统实现中仍可能存在缺陷[17]. 传统的形式化验证方法难以找出这些系统实现中的缺陷. 近年来, 也有部分研究者考虑使用形式化规约来对分布式系统实现进行测试. 如 MongoDB 开发者 Davis 等人尝试利用 TLA+对 MongoDB 数据库中一个数组存取算法进行建模[44], 并生成输入输出作为测试用例. 但这样的方法无法支持测试真实分布式系统广泛存在的不确定性, 如消息通讯和故障. 因此该方法实际上无法检测到第 3.2 节中所描述的所有缺陷.

实现级模型检查: 实现级模型检查器[21−25]通过对分布式系统代码进行插桩, 并在运行时对系统中的网络消息进行拦截和乱序, 以使分布式系统进入普通测试难以抵达的边缘状态, 进而发现隐藏在系统实现中的深层缺陷(deep bug). 然而, 实现级模型检查方法缺少有效的测试预言, 且由于直接作用于分布式系统实现中, 面临着严重的状态空间爆炸问题.

面向大型软件的基于模型的测试方法: 基于模型的测试是一个成熟且被广泛研究的领域[45]. 这些测试方法通过对目标程序进行建模, 并根据模型生成测试用例以测试程序的特定行为或性质. Bishop 等人为 TCP 和 UDP 通信开发了形式化规约, 并测试了其在 FreeBSD、Linux 和 WinXP 这 3 个操作系统中的实现[46]. Li 等人利用 Coq 对网络应用中的数据包延迟行为进行了建模, 并对网络服务器 Apache 及 Nginx 进行了测试[47]. Kim 等人针对分布式存储系统的最终一致性, 即各节点上的数据最终会达成收敛, 提出了差异重同步模型, 以检测分布式系统中的收敛失败缺陷[48]. 这些方法仅可用于测试特定的程序行为或性质, 而无法应对分布式系统中庞大的状态空间, 因此测试完备性较低.

面向 Android 应用的基于状态图的测试技术: 基于状态图生成测试用例的技术被广泛应用在 Android 应用测试领域. 这类技术通过构建 Android 应用的状态空间图并进行图遍历以生成测试用例. 其中 Dynodroid[49]、GUICC[50]使用随机策略对状态图进行遍历. A3E[51]、DroidBot[52]使用深度优先方法对状态图进行遍历. Stoat[53]、CrawDroid[54]使用基于权重的方法对状态图进行遍历. 这些方法仅面向状态数量少于 100 的小规模状态图. Fastbot[55]结合多种探索策略如 UCB 算法、MTree 算法、Q-learning 算法来探索大规模的状态空间. 该方法可能会是我们测试方法的有效补充, 我们将在未来工作中评估是否可以在生成测试用例时使用类似的组合策略.

面向分布式系统的缺陷检测方法: 近年来研究者进行了大量针对分布式系统的缺陷检测研究. Liu 等人提出了 DCatch[56]与 FCatch[57]分别用于检测分布式系统的中的并发缺陷以及故障时机缺陷. Lu 等人提出的 CrashTuner[58]可以检测分布式系统中的失效恢复缺陷. Chen 等人提出了 CoFI[59]用于检测分布式系统的网络分区相关缺

陷. 这些工作使用特定的模式来检测特定种类的缺陷, 并且不能被应用于检测其他缺陷类型.

故障注入测试: 向分布式系统中主动注入故障来暴露缺陷是一种常用的测试技术. 工业界使用 Chaos Monkey[60] 和 Jepsen[61]等工具通过向目标系统中随机注入节点失效等各类型故障来触发隐藏在分布式系统内的缺陷. Gunawi 等人提出了 Fate[62]框架来对分布式系统的故障测试场景进行定制. Joshi 等人在此基础上进一步提出了 PreFail[63]框架来帮助测试者定制故障注入策略, 以减少待测试的状态空间. Alquraan 等人提出了 NEAT[64]来向分布式系统中注入网络分区故障. Alagappan 等人提出了 PACE[65]来针对分布式系统中的级联崩溃相关缺陷进行测试. 这些测试方法主要关注对故障的注入模拟, 与本文所提出的测试方法相比缺乏用于检测缺陷的准确预言.

6 总 结

我们提出了基于 TLA+形式化规约的测试方法来对分布式系统进行系统化测试, 填补形式化规约与实现源码之间的鸿沟. 给定目标系统及 TLA+规约, 该方法能够系统地验证其系统实现是否遵从了规约中的系统设计. 我们将该方法应用到 Raft 协议及两个 Raft 系统实现中, 并成功发现了 3 个未知缺陷和 2 个已知缺陷, 证明该方法是可行且有效的.

References:

[1] TiDB. 2017. https://github.com/pingcap/tidb

[2] CockroachDB. 2020. https://github.com/cockroachdb/cockroach

[3] Hunt P, Konar M, Junqueira FP, Reed B. ZooKeeper: Wait-free coordination for Internet-scale systems. In: Proc. of the 2010 USENIX Conf. on USENIX Annual Technical Conf. Boston: USENIX Association, 2010. 145–158.

[4] Ongaro D. Consensus: Bridging theory and practice [Ph.D. Thesis]. Stanford: Stanford University, 2014.

[5] Ethereum. 2022. https://ethereum.org/en/

[6] The home of the fabric mod development toolchain. 2020. https://fabricmc.net/

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

[8] 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.

[9] Cao W, Liu ZJ, Wang P, Chen S, Zhu CF, Zheng S, Wang YH, Ma GQ. PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database. Proc. of the VLDB Endowment, 2018, 11(12): 1849–1862. [doi: 10.14778/3229863.3229872]

[10] Raft-Java. 2017. https://github.com/wenweihu86/raft-java

[11] Xraft. 2018. https://github.com/xnnyygn/xraft

[12] Lamport L. The TLA+ home page. 1999. https://lamport.azurewebsites.net/tla/tla.html

[13] TLA+ specification for the Raft consensus algorithm. 2014. https://github.com/ongardie/raft.tla

[14] Newcombe C, Rath T, Zhang F, Munteanu B, Brooker M, Deardeuff M. How Amazon Web services uses formal methods. Communications of the ACM, 2015, 58(4): 66–73. [doi: 10.1145/2699417]

[15] Foundations of azure cosmos DB (multi-master) with Dr. Leslie Lamport. 2018. https://www.youtube.com/watch?v=kYX6UrY_ooA

[16] Gu XS, Wei HF, Qiao L, Huang Y. Raft with out-of-order executions. Ruan Jian Xue Bao/Journal of Software, 2021, 32(6): 1748–1778 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/6248.htm [doi: 10.13328/j.cnki.jos.006248]

[17] 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]

[18] 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]

[19] Wilcox JR, Woos D, Panchekha P, Tatlock Z, Wang X, Ernst MD, Anderson T. Verdi: A framework for implementing and formally verifying distributed systems. In: Proc. of the 36th Annual ACM SIGPLAN Conf. on Programming Language Design and Implementation. Portland: ACM, 2015. 357–368. [doi: 10.1145/2737924.2737958]

[20] Michael E, Woos D, Anderson T, Ernst MD, Tatlock Z. Teaching rigorous distributed systems with efficient model checking. In: Proc. of the 14th EuroSys Conf. Dresden: ACM, 2019. 32. [doi: 10.1145/3302424.3303947]

[21] 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.

[22] 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.

[23] Yabandeh M, Knežević N, Kostić D, Kuncak V. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In: Proc. of the 6th USENIX Symp. on Networked Systems Design and Implementation. Boston: USENIX Association, 2009. 229–244.

[24] 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.

[25] 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 EuroSys Conf. Dresden: ACM, 2019. 20. [doi: 10.1145/3302424.3303986]

[26] The TLA+ toolbox. 2010. http://lamport.azurewebsites.net/tla/toolbox.html

[27] ASM. 2023. https://asm.ow2.io

[28] Python NetworkX. 2023. https://networkx.org

[29] The Raft consensus algorithm. 2014. https://raft.github.io

[30] Hazelcast. 2008. https://github.com/hazelcast/hazelcast

[31] SOFAJRaft. 2019. https://github.com/sofastack/sofa-jraft

[32] Zhao C. The Practice on Developing the Distributed Consensus Algorithm. Beijing: Peking University Press, 2020 (in Chinese).

[33] Xraft issue: Duplicate vote response can make illegal leader without a Quorum. 2022. https://github.com/xnnyygn/xraft/issues/27

[34] Xraft commit: Handle with canceled votes. 2022. https://github.com/xnnyygn/xraft/pull/28/commits/a48000080b6590402fbf45dd1a06af001d558830

[35] Xraft issue: Votedfor is not stored when a node is candidate and receives an appendentriesrpc. 2022. https://github.com/xnnyygn/xraft/issues/29

[36] Raft-Java issue#3. 2017. https://github.com/wenweihu86/raft-java/issues/3

[37] Raft-Java issue#19. 2019. https://github.com/wenweihu86/raft-java/issues/19

[38] The Coq proof assistant. 1984. https://coq.inria.fr/

[39] Gomes VBF, Kleppmann M, Mulligan DP, Beresford AR. Verifying strong eventual consistency in distributed systems. Proc. of the ACM on Programming Languages, 2017, 1: 109. [doi: 10.1145/3133933]

[40] Lesani M, Bell CJ, Chlipala A. Chapar: Certified causally consistent distributed key-value stores. ACM SIGPLAN Notices, 2016, 51(1): 357–370. [doi: 10.1145/2914770.2837622]

[41] Sergey I, Wilcox JR, Tatlock Z. Programming and proving with distributed protocols. Proc. of the ACM on Programming Languages, 2018, 2: 28. [doi: 10.1145/3158116]

[42] Deligiannis P, Donaldson AF, Ketema J, Lal A, Thomson P. Asynchronous programming, analysis and testing with state machines. ACM SIGPLAN Notices, 2015, 50(6): 154–164. [doi: 10.1145/2813885.2737996]

[43] Ji Y, Wei HF, Huang Y, Lü J. Specifying and verifying CRDT protocols using TLA+. Ruan Jian Xue Bao/Journal of Software, 2020, 31(5): 1332–1352 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5956.htm [doi: 10.13328/j.cnki.jos.005956]

[44] Davis AJJ, Hirschhorn M, Schvimer J. Extreme modelling in practice. Proc. of the VLDB Endowment, 2020, 13(9): 1346–1358. [doi: 10.14778/3397230.3397233]

[45] Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 2013, 86(8): 1978–2001. [doi: 10.1016/j.jss.2013.02.061]

[46] Bishop S, Fairbairn M, Mehnert H, Norrish M, Ridge T, Sewell P, Smith M, Wansbrough K. Engineering with logic: Rigorous test-oracle specification and validation for TCP/IP and the sockets API. Journal of the ACM, 2018, 66(1): 1–77. [doi: 10.1145/3243650]

[47] Li YS, Pierce BC, Zdancewic S. Model-based testing of networked applications. In: Proc. of the 30th ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Virtual: ACM, 2021. 529–539. [doi: 10.1145/3460319.3464798]

[48] Kim BH, Kim T, Lie D. Modulo: Finding convergence failure bugs in distributed systems with divergence resync models. Proc. of the 2022 USENIX Annual Technical Conf. Carlsbad: USENIX Association, 2022. 383–398.

[49] Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android APPs. In: Proc. of the 9th Joint Meeting on Foundations of Software Engineering. Saint Petersburg: ACM, 2013. 224–234. [doi: 10.1145/2491411.2491450]

[50] Baek YM, Bae DH. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In: Proc. of the 31st

IEEE/ACM Int'l Conf. on Automated Software Engineering. Singapore: ACM, 2016. 238–249. [doi: 10.1145/2970276.2970313]

[51] Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android APPs. ACM SIGPLAN Notices, 2013, 48(10): 641–660. [doi: 10.1145/2544173.2509549]

[52] Li YC, Yang ZY, Guo Y, Chen XQ. DroidBot: A lightweight UI-guided test input generator for Android. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Software Engineering Companion. Buenos: IEEE, 2017. 23–26. [doi: 10.1109/ICSE-C.2017.8]

[53] Su T, Meng GZ, Chen YT, Wu K, Yang WM, Yao Y, Pu GG, Liu Y, Su ZD. Guided, stochastic model-based GUI testing of Android APPs. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. Paderborn: ACM, 2017. 245–256. [doi: 10.1145/3106237.3106298]

[54] Cao YZ, Wu GQ, Chen W, Wei J. CrawlDroid: Effective model-based GUI testing of Android APPs. In: Proc. of the 10th Asia-Pacific Symp. on Internetware. Beijing: ACM, 2018. 19. [doi: 10.1145/3275219.3275238]

[55] Cai TQ, Zhang Z, Yang P. Fastbot: A multi-agent model-based test generation system: Beijing Bytedance Network Technology Co. Ltd. In: Proc. of the 15th IEEE/ACM Int'l Conf. on Automation of Software Test (AST). Seoul: IEEE, 2020. 93–96.

[56] Liu HP, Li GP, Lukman JF, Li JX, Lu S, Gunawi HS, Tian C. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. ACM SIGPLAN Notices, 2017, 52(4): 677–691. [doi: 10.1145/3093336.3037735]

[57] 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]

[58] 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]

[59] 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. Virtual Event, Australia: ACM, 2020. 536–547. [doi: 10.1145/3324884.3416548]

[60] ChaosMonkey. 2023. https://github.com/Netflix/chaosmonkey

[61] Jepsen. 2022. https://jepsen.io/

[62] Gunawi HS, Do T, Joshi P, Alvaro P, Hellerstein JM, Arpaci-Dusseau AC, Arpaci-Dusseau RH, Sen K, Borthakur D. FATE and DESTINI: A framework for cloud recovery testing. In: Proc. of the 8th USENIX Conf. on Networked Systems Design and Implementation. Boston: USENIX Association, 2011. 238–252.

[63] Joshi P, Gunawi HS, Sen K. PREFAIL: A programmable tool for multiple-failure injection. ACM SIGPLAN Notices, 2011, 46(10): 171–188. [doi: 10.1145/2076021.2048082]

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

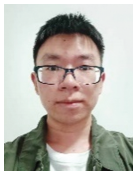
[65] Alagappan R, Ganesan A, Patel Y, Pillai TS, Arpaci-Dusseau AC, Arpaci-Dusseau RH. Correlated crash vulnerabilities. In: Proc. of the 12th USENIX Conf. on Operating Systems Design and Implementation. Savannah: USENIX Association, 2016. 151–167.

附中文参考文献:

[16] 谷晓松, 魏恒峰, 乔磊, 黄宇. 支持乱序执行的 Raft 协议. 软件学报, 2021, 32(6): 1748–1778. http://www.jos.org.cn/1000-9825/6248.htm [doi: 10.13328/j.cnki.jos.006248]

[32] 赵辰. 分布式一致性算法开发实战. 北京: 北京大学出版社, 2020.

[43] 纪业, 魏恒峰, 黄宇, 吕建. CRDT 协议的 TLA+描述与验证. 软件学报, 2020, 31(5): 1332–1352. http://www.jos.org.cn/1000-9825/5956.htm [doi: 10.13328/j.cnki.jos.005956]



王栋(1996－), 男, 博士生, CCF 学生会员, 主要研究领域为分布式系统测试, 形式化方法.

窦文生(1984－), 男, 博士, 研究员, 博士生导师, CCF 专业会员, 主要研究领域为软件工程, 分布式系统测试, 数据库测试.

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

吴陈傲(1998－), 男, 硕士, 主要研究领域为形式化方法.

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.

黄涛(1965－), 男, 博士, 研究员, 博士生导师, CCF 高级会员, 主要研究领域为网络分布式计算, 软件工程.