南京大學
NANJING UNIVERSITY

研 究 生 毕 业 论 文

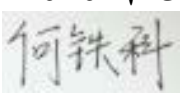
（申请硕士专业学位）

论 文 题 目	基于故障注入的以太坊私有链性能测试系统的设计与实现
作 者 姓 名	王新宇
专 业 名 称	工程硕士（软件工程领域）
研 究 方 向	软件工程
指 导 教 师	刘嘉 副教授 何铁科 助理研究员

2020 年 4 月 18 日

学　　号：**MF1832165**

论文答辩日期：**2020 年 5 月 23 日**

指 导 教 师：  **（签字）**

The Design and Implementation of Performance Testing System for Private Ethereum Blockchain via Fault Injection

By

Xinyu Wang

Supervised by

Associate Professor **Jia Liu**

Assistant Research Fellow **Tieke He**

A Thesis

Submitted to the Software Institute

and the Graduate School

of Nanjing University

in Partial Fulfillment of the Requirements

for the Degree of

Master of Engineering

Software Institute

May 2020

南京大学研究生毕业论文中文摘要首页用纸

毕业论文题目：基于故障注入的以太坊私有链性能测试系统的设计与实现

软件工程 专业 2020 级硕士生姓名：王新宇

指导教师（姓名、职称）：刘嘉 副教授 何铁科 助理研究员

摘 要

区块链技术近年来备受关注，它为智能合约提供了必要的可信环境，使得智能合约在区块链平台上广泛运用。以太坊是目前最知名的智能合约支撑平台，它支持私有链的搭建，但是其性能受规模和用户数量等因素的制约，需要进行性能评估。以太坊的节点共识、通信需要大量计算、网络资源，导致性能不稳定。同时，区块链常部署在不稳定的环境中，可能面临不可预测的故障。因此，在实际应用中，应模拟不稳定的部署环境并对以太坊进行充分的性能测试。

本文对以太坊的性能进行深入研究，从以太坊的 PoW 共识协议与 Gas 机制入手，分析共识流程与 Gas 机制对以太坊挖矿速度与区块容量的影响，最终提出了两个以太坊私有链的性能影响因素：Difficulty、Gas Limit。本文引入故障注入技术，通过对混沌工程领域中提出的分布式系统常见故障进行分析，结合以太坊共识流程与智能合约执行原理，提出了以太坊私有链中的四种故障类型：应用、共识、智能合约、网络，并根据故障严重程度进行等级划分。据此，本文实现了基于故障注入的以太坊私有链性能测试系统。本系统根据功能划分为两个模块，测试管理模块与测试执行模块。测试管理模块实现配置管理与结果生成功能。该模块使用 Django 服务器实现，MongoDB 作为数据库，并使用 Python 对数据结果进行处理。测试执行模块实现测试链搭建、性能测试与故障注入功能。测试链搭建使用 Docker 技术，实现全自动快速搭建。性能测试选用异步操作友好的 NodeJS，实现多线程请求发送，并记录秒级指标数据。故障注入使用线程通信与操作 Docker 容器实现，通过抽象接口实现良好的可扩展性。

本系统可以对以太坊私有链进行快捷有效的性能测试，对以太坊性能进行系统评估。通过故障注入技术，可以在测试环境下模拟真实场景下的故障，获取更贴近真实场景的测试结果。本文使用本系统进行了两组实验，即性能影响因素验证实验与故障注入验证实验。影响因素验证实验中，随着影响因素的变化，以太坊吞吐量下降，延迟大幅增长。故障注入验证实验中，在执行四类故障注入的时间段内，以太坊性能均体现出不同程度的下降趋势。实验结果证明，本系统可以很好地通过性能指标展示以太坊在故障影响下的性能与稳定性。

关键词：区块链，以太坊，性能测试，故障注入

南京大学研究生毕业论文英文摘要首页用纸

THESIS: The Design and Implementation of Performance Testing System for Private Ethereum Blockchain via Fault Injection

SPECIALIZATION: Software Engineering

POSTGRADUATE: Xinyu Wang

MENTOR: Associate Professor **Jia Liu**
Assistant Research Fellow **Tieke He**

Abstract

Blockchain gains a lot of attention in recent years. It provides the necessary trusted environment for smart contracts, making smart contracts widely used on blockchain platforms. Ethereum is currently the most well-established smart contract support platform. Ethereum supports private blockchains, but its performance is affected by the blockchain configuration and number of nodes. Thus performance evaluation is required. Ethereum needs massive calculations and network resources to reach consensus, causing instability. Meanwhile, blockchains are often deployed in unstable environments in which blockchains may face unpredictable faults. Therefore, for adequate performance and stability testing, private Ethereum blockchain should be tested in unstable environment.

In this paper, we conducted in-depth research on the performance of Ethereum. By analyzing Ethereum's PoW consensus protocol and Gas mechanism, we finally recognized two performance impact factors for private Ethereum blockchain: Difficulty, Gas Limit. Also, we introduce fault injection technology. By analyzing the common faults of distributed systems along with blockchain characteristics, we propose four types of faults on private Ethereum blockchain: application, consensus, smart contracts, network. Afterwards, we divide each failure into three levels according to the severity of the fault. Accordingly, we implement a performance testing System for private Ethereum blockchain via fault injection. The system is divided into two modules according to system features, test management module and test execution module. The test management module implements configuration management and result generation.

This module is implemented with a Django server, MongoDB as a database. Python is used to process the data results. The test execution module implements test chain construction, performance testing and fault injection. Docker technology is used to achieve fully automatic and rapid blockchain construction. This module is basically implemented by NodeJS, which has good support for asynchronous operations. Fault injection is implemented by thread communication and operating Docker containers.

The system can perform rapid and effective performance tests on private Ethereum blockchain and evaluate the performance of Ethereum adequately. Through fault injection technology, real-world faults can be simulated in the test environment. Thus test results are closer to real scenarios. We design and perform two sets of experiments, including impact factor verification experiments and fault injection verification experiments. In the impact factor verification experiment, with the change of the factor, the throughput decreased and the latency increased significantly. In the fault injection verification experiment, the performance of Ethereum showed a downward trend during fault injection. The experimental results showed that the system can reflect the performance and stability of Ethereum under fault injection through performance indicators.

Keywords: Blockchain, Ethereum, Performance Testing, Fault Injection

目录

表 目 录

图 目 录

第一章　引言

1.1　项目背景及意义

二十一世纪初期，人工智能、数据挖掘、区块链等新技术也相继涌现。区块链源于 2008 年中本聪提出的比特币 [1]，它是一种分布式系统 [2]，节点间通过共识协议相互通讯。区块链的去中心化、不可篡改的特性有效降低了交易成本，使人们得以在无人监管的模式下构建可信交易平台 [3]。因此，区块链技术有很高的应用价值，可用领域十分宽泛，如物联网 [4]、信息安全 [5] 等。智能合约是一种信息化的计算机协议，无需人来监督执行 [6]。一旦达成合约中的条件，合约会自动执行，这有效地降低了违约的风险。区块链去中心化、不可篡改的特性为实施智能合约提供了必要的可信环境。以太坊是首个支持图灵完备的智能合约编程语言的区块链平台 [7]。以太坊支持私有链，私有链是一种可以根据个人需求进行部署的区块链，具有可配置、规模小、需要许可等特点。这使得以太坊可以更好的满足企业和个人的使用需求。

区块链去中心化的特性、复杂的共识流程导致其吞吐量低、交易延迟高、性能差。目前，人们常用的支付宝、微信以及 Visa 等支付方式早已突破 10000 笔/秒，而以太坊公有链平均每秒钟处理 15 笔交易，每笔交易需要等待 3 分钟的确认时间 [8]，很难满足当今的数字支付场景 [9]。私有链是一种良好的解决方案，它采用许可机制，可以限制用户规模，降低通信开销，进而提升性能 [10]。以太坊私有链是目前较为成熟的私有链平台，用户可以根据自身需求配置私有链。以太坊私有链根据启动配置与部署规模不同，可能有不同的性能表现 [11]。由于用户使用以太坊私有链的场景对性能可能有要求，需要进行性能评估。因此，在实际应用中，应仔细考虑以太坊私有链的性能并进行充分测试。

分布式系统的真实部署环境不稳定，可能会面临各种类型的故障 [12]。性能测试往往在没有外界干扰的测试环境中执行，这种环境相对真实环境更稳定，发生故障的概率更低，但是难以反映真实场景中的结果。作为一种分布式系统，区块链也面临同样的问题，其稳定性需要得到保障。区块链在真实环境中总是面临不可预测的故障 [13]，这可能导致性能下降，甚至整个系统崩溃 [14]。由于区块链使用复杂的共识协议，需要更多的通信与资源开销，这使得发生故障可能对区块链系统产生更严重的影响。因此，开发人员需要验证以太坊私有链能否在故障条件下满足用户的需求。如果他们想要获得更接近真实场景的性能测试结果，则必须在测试过程中模拟真实场景中的故障。

根据上述分析，本文实现了基于故障注入的以太坊性能测试系统，可以尽可能暴露性能问题，帮助以太坊私有链维护人员评估系统的性能与可靠性。本文中依次实现了：（1）通过对以太坊私有链的性能分析，提出两个以太坊性能影响因素，Difficulty、Gas Limit；（2）通过对分布式系统故障与区块链的特性进行分析，提出四种类型的故障，即应用故障、共识故障、智能合约故障、网络故障。本系统可以获取以太坊私有链的秒级性能指标信息，从而对以太坊的性能进行准确评估。本系统引入故障注入技术，可以在测试环境中注入故障，探究在故障情况下以太坊私有链的稳定性。最终，本系统可以对整个以太坊私有链进行一个完整的系统的评估。

1.2　国内外研究现状

本文使用了性能测试技术，对以太坊私有链性能进行评估，并引入了故障注入技术，在测试环境中模拟真实场景下的故障，获取更真实的测试结果。本节介绍了与本系统研究内容相关技术的国内外研究现状，从性能测试与故障注入两个方面分别展开介绍。

1.2.1　性能测试

性能测试是通过人为模拟真实场景下的负载，对系统进行测试以评估其性能的一种手段。性能测试通过对被测系统加压，验证被测系统在正常、峰值以及异常的负载压力下的性能表现，进而找出其潜在的性能瓶颈 [15]。性能测试工具往往通过模拟用户请求并批量发送的方式达到模拟真实系统的负载的目的。对于传统软件系统如 Web 应用的性能测试，目前已有较为成熟的工具。如 Micro Focus 发布的性能测试工具 LoadRunner，用于测量负载下的系统行为和性能，可对企业级大规模云服务进行性能测试 [16]。JMeter 是 Apache 基金会开源的性能测试工具，由 Java 语言实现，可以对待测系统进行功能与性能测试。JMeter 最初是为 Web 应用设计的性能测试工具，但由于其良好的设计使其支持以插件的形式扩展，使其具备根据用户需求对不同形式的软件系统进行测试的能力 [17]。国内著名云服务公司阿里云也发布了性能测试服务 PTS（Performance Testing Service）[18]。PTS 是卓越的 SaaS 性能测试平台，具备强大的分布式压测能力，可模拟海量用户的真实业务场景。

传统分布式性能测试方法通过模拟性能相关用例场景，映射到实际部署场景，向节点批量发送请求，获取请求返回信息生成测试报告 [19]。区块链作为分布式系统，其性能测试方法与传统分布式性能测试方法类似。目前，Dinh 等人研究了常规私有链性能测试方法 [20]。他们关注于共识和智能合约执行等区块

链层级对区块链的影响，通过性能指标（例如吞吐量和延迟）评估影响程度，并实现了 Blockbench 工具以测试区块链性能。Zheng 等人提出了一种基于日志解析的以太坊区块链性能监控框架 [21]。这种框架可以获取更详细的信息，并且不会影响区块链本身的性能。2018 年华为公司发布了首个区块链专用的性能测试工具 Hyperledger Caliper[22]。Caliper 是一个区块链性能基准框架，允许用户使用预定义的用例测试不同的区块链解决方案。目前该工具已经被超级账本项目（Hyperledger）收录为超级账本系列区块链系统官方性能测试工具。

目前，对传统软件系统的性能测试已较为成熟，而在私有区块链性能测试领域的研究较少。并且已有研究局限于对某一特定类型的区块链做整体的性能评估，缺少对区块链系统运行过程中不同时间节点的性能指标计算，难以分析对区块链系统上发生的事件对性能产生的影响。

1.2.2　故障注入

故障注入是一种软件测试技术，用于验证软件的可靠性。该技术通过向系统中注入故障模拟真实场景。通过观察系统中存在故障时的行为，可以分析系统的稳定性 [23]。一个完整的故障注入流程需要首先需要准备故障注入环境，并让待测系统以常规负载运转。故障注入过程中，需要选择一种故障类型对目标系统进行故障注入，同时需要对系统进行监控以确认故障带来的影响 [24]。

故障注入技术最先用于传统软件系统，但由于传统软件多为集中式系统，规模小且缺陷容易修复，因此该技术未被广泛使用。在分布式系统中，系统组成复杂节点之间相互依赖，导致缺陷不易暴露与修复 [25]。这种情况下，故障注入可以有效暴露缺陷，因此在分布式领域中，该技术得到充分运用。Ziade 等人对软件系统故障进行了分类，并对故障注入方法进行了总结。他认为，一个故障注入的实现需要故障注入器、工作负载生成、数据收集与分析器等组件 [24]。江苏大学的陈教授分析了静态与动态两种故障注入方法，对已有故障注入工具进行了实验验证，最终总结了故障注入技术存在的问题 [26]。Yuan 等人通过对 198 个现实世界中发生在分布式的故障事件进行实证研究 [27]，发现分布式系统故障往往是由极少部分的节点故障导致，并且通过有效的故障注入手段可以避免类似的故障出现。随着故障注入技术的成熟，研究人员逐渐将这种技术工程化。Netflix 工程师提出了混沌工程（Chaos Engineering）的概念 [14]——一种通过实验来理解具有复杂行为和故障模式的分布式系统的方法。混沌工程概念将故障注入技术工程化，并提出五项混沌工程原则，并以此为基准设计故障注入实验。Alvaro 等人认为大型分布式系统的构建必须要预见故障的出现，减轻故障影响

[28]。他们以 Netflix 的系统为研究对象，提出一种“迭代驱动故障注入”（LDFI）的技术，并描述了该技术如何适应大规模系统的复杂且动态的现状。

故障注入技术在企业中也受到广泛关注，在国外由 Netflix 公司牵头推动与发展故障注入技术。2011 年，Netflix 发布了首个故障注入工具 Chaos Monkey，并以此为基础实现了 FIT 平台（Failure Injection Testing）[28]。在注入故障的同时，FIT 还能以可控的方式在整个 Netflix 生态系统上传播故障。由于该平台是专门为 Netflix 公司内部故障测试而开发的，所有具有一定的局限性。阿里巴巴开源了内部的混沌工程工具 ChaosBlade[29]。作为国内首个推出的混沌工程工具，ChaosBlade 支持多种类型的故障，包括节点宕机、网络延迟等。同时，ChaosBlade 也支持多种故障注入的对象，如 Docker、Kubernates、JVM 等。该工具目前已在公司内部广泛使用，并取得一定的成效。

1.3　本文的主要工作

本文通过对以太坊私有链的研究，分析得出以太坊私有链系统的性能影响因素 Diffculty 与 Gas Limit，并通过获取区块链系统常用的性能指标对以太坊私有链的性能进行准确评估。为了探究以太坊私有链的稳定性，本文引入故障注入技术，根据区块链四个层次的角度分别提出四种类型的故障，即应用故障、共识故障、智能合约故障、网络故障，并根据故障的严重等级为每种故障划分了三个等级。使用故障注入技术向以太坊私有链注入这四种故障，记录故障注入时与故障恢复后的性能表现，探究以太坊私有链的性能与稳定性。

在上述方法的基础上，本文实现了一种基于故障注入的以太坊私有链性能测试系统。首先，根据对以太坊性能与稳定性的分析，实现了一个支持区块链自动部署、批量请求发送、可配置故障注入与测试结果收集的以太坊私有链性能测试工具。其次，在此测试工具的基础上，搭建了一套测试集生成、测试结果存储与可视化的平台。最后，针对每种故障的每个等级分别进行了故障注入实验。实验结果表明：（1）系统可以测量以太坊私有链的两个性能指标（即吞吐量和交易延迟）；（2）实验结果可以验证两个性能影响因素和四种故障对性能与稳定性的影响。

1.4　本文的组织结构

本文总共分为六个章节，组织结构如下：

第一章，引言部分。介绍项目背景与意义，分析当前区块链系统的性能现状，介绍当前国内外对性能测试、故障注入与区块链的研究以及本文的工作。

第二章，相关概念与技术。介绍与项目相关的概念与技术，包括以太坊区块链、性能测试技术与故障注入技术。

第三章，需求分析与概要设计。首先进行需求分析，明确了项目的涉众与用例。在此基础上，确定了功能与非功能需求。然后对项目进行了概要设计，从整体的角度给出系统的架构与 4+1 视图。针对测试用例与测试结果存储给出了数据持久化模型。

第四章，详细设计与实现。在概要设计的基础上，介绍了测试管理模块与测试执行模块及其子模块的详细设计与实现，给出了子模块设计图和时序图，并对于每个子模块核心部分展示了部分代码。

第五章，实验评估与分析。使用研发完成的系统，对以太坊私有链进行测试。设计了多种不同类型的测试用例，对每种故障类型分别进行了测试，并对测试结果进行研究分析。

第六章，总结与展望。总结完成项目与论文期间所做的工作，展望项目的后续研究工作。

1.5　本章小结

本章主要描述了本文所做的主要研究的背景与意义，提出了本项目的动机，深入分析了本项目的研究价值。之后，介绍了与本文相关的研究内容的国内外研究现状，为后续研究做铺垫。最后，介绍了本文的主要工作与组织结构。

第二章　相关概念与技术

本系统为一种基于故障注入的以太坊私有链性能测试系统，是一个为以太坊用户提供高效的性能测试服务的系统，其测试对象为以太坊私有链。从与本系统相关的三类技术：区块链、性能测试、故障注入，结合本文的研究内容，对相应概念与技术进行详细介绍。

2.1　区块链

区块链作为第一种可行的去中心化技术，近年来备受关注。区块链本质上是一种数据结构，起源于中本聪发明的比特币。由于比特币系统连续 10 年稳定运行而且具有去中心化、无法篡改的特性，使其逐渐受到人们的关注。目前，比特币的市值已经超越 10000 亿人民币 [30]。比特币白皮书中阐述了比特币所使用的 P2P 网络技术、工作量证明协议等 [1]，为区块链技术的出现奠定了基础。开发者从比特币的底层实现中抽取出了数据存储部分。由于其数据类似一个个区块收尾相连，因此命名为区块链技术。2014 年，以太坊 [7] 出现，它引入了智能合约的概念，智能合约由图灵完备的编程语言实现，使得区块链能实现更加灵活且复杂的功能。

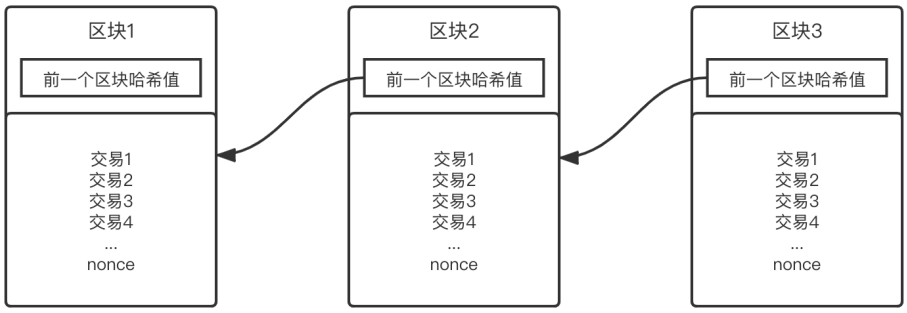


图 2.1: 区块链结构示意图

如图2.1所示，区块链是一条按照时间顺序生成的区块链接在一起形成的链。每个区块中都包含上一个区块的哈希值，并通过自身内容与上个区块的哈希值计算本区块的哈希值。如任何人想要改变区块链中的某一个值，由于哈希函数的特性，该区块的哈希值发生变化，并且在该区块之后生成的区块哈希值均会

变化，需要将之后所有的区块的哈希值按照共识协议的规定重新计算，这往往是不可行的。因此，区块链中的数据难以篡改。

区块链根据权限可分为公有链、联盟链、私有链。联盟链与私有链都需要一定的权限，因此也统称为需要许可的区块链。公有链是指任何人都可以随时读取区块数据、发送可确认交易、竞争打包区块的区块链。公有链往往只有一条，只有这条链上的交易才被世界承认。需要许可的区块链则指由一个或多个组织或机构管理的区块链，每个机构都运行着一个或多个节点，只允许内部组织机构或经他们授权的用户读取和发送交易，并且共同来记录交易数据。私有链相比联盟链规模更小，对权限的要求更高。与公共链不同，任何人都可以用自己的配置来启动私有链，并且私有链之间的交易都是独立的。私有链的性能与稳定性表现与公有链有显著差异 [31]。公有链由于节点数量多、通信开销大而产生性能瓶颈。私有链用户与节点远少于公共链，但同时负责打包区块的节点也更少，私有链根据启动配置与节点规模不同体现出不同的性能表现。

2.1.1　共识协议

区块链技术具有去中心化等特点。这种技术不需要一个中心化的管理者来管理这条区块链，而是每个人拥有均等的权限，每个人都有机会去生成下一个区块，每个人都有权力去验证其他人生成的区块。区块链技术采用共识协议的方式使所有人对区块达成一致，在某一节点挖出区块后，每一个节点根据事先规定的共识协议验证区块，验证通过后区块会被真正记录到区块链中 [32]。

共识协议通过预先设定好的协议使区块链上所有节点对区块的内容达成一致。这种协议与传统分布式系统的共识选举协议不同。传统分布式系统的共识协议多为 Master/Slave 架构中，通过选举选出新的 Master 节点，由 Master 节点进行统一共识。这种协议只能解决节点故障问题，但是非可信环境下的节点作恶问题无法解决，无法保证信息的安全性 [33]。由于区块链往往部署在非可信环境，这种环境下可能出现二次支付、自私挖矿等节点作恶行为，区块链中的共识协议必须能够识别并防止以上现象发生 [34]。

目前常见的区块链共识协议有工作量证明（PoW）、拜占庭容错（BFT）、权益证明（PoS）等 [35]。工作量证明，又称 PoW 协议，是比特币发起者中本聪提出的一种共识协议。这种共识协议可以在不安全的环境下保证所有节点达成共识。PoW 协议通过解决并验证一个预先约定好的难题达成共识，解决该难题需要一定的代价，比如需要大量算力计算一定的时间。图2.2给出了 PoW 协议的示意图。在诸多挖矿的竞争者中，节点 A 通过大量计算成功解决了难题，并成功

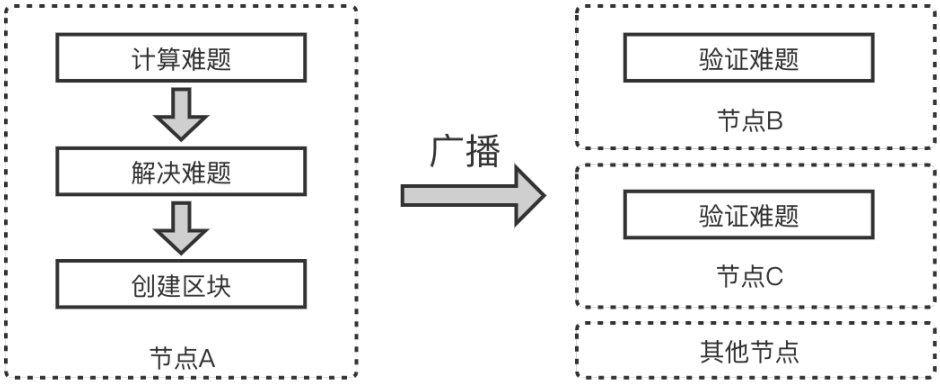


图 2.2: 工作量证明共识协议示意图

打包区块。之后，节点 A 将难题结果置于区块中广播出去，其他节点收到该区块的信息，验证其中的难题的解是否合法，如果合法则将该区块链接到当前区块链的末尾。在比特币中，这个难题是通过哈希函数达成一个特定条件，即计算出一个随机值，使该值与交易信息合并到一起计算出的哈希值满足前 n 位都是 0 的条件（n 取决于挖矿难度）。当某一节点成功达成该条件并广播该区块信息，其他节点拿到区块信息后计算哈希值。如果哈希值满足该特殊条件，则认定该区块合法，并连接到区块链末尾。根据哈希函数具有碰撞抗性，即任意微小的变化均会导致哈希结果大幅度变化 [36]，矿工只能通过不断尝试哈希计算得到该随机值，因此需要大量的算力。

除 PoW 协议外，BFT 与 PoS 也是常见的区块链共识协议。BFT（Byzantine Fault Tolerance），又称拜占庭容错，通过所有节点之间反复通信确认来达成共识，这种共识协议不需要大量计算资源，但是对网络通信开销要求巨大，而且系统的通信开销随着节点的增长呈指数级增长 [37]。因此，使用此类共识协议的区块链往往不能有过多的节点，往往用于私有链 [38]。PoS（Proof of Stake），又称权益证明，是一种以在链上所有的价值作为判断打包区块权力归属的共识协议。以所有矿工的所有资产的比例作为权重，随机选取矿工进行打包，如果矿工作恶，则将其所有资产没收。这种共识协议不耗费网络与计算资源，但是容易造成资产集中，最终造成打包权的垄断 [39]。

2.1.2　智能合约

智能合约是一种数字化的计算机协议，由 Nick Szabo 于 1995 年首次提出[6]。起初区块链技术只适用于交易转账，功能简单导致使用场景单一。智能合约由图灵完备的编程语言实现，可编程的交易将区块链从单纯的转账拓展到复杂的使用场景。通过支持智能合约，区块链技术进入 2.0 时代 [40]。2014 年，以太坊黄皮书发布，支持图灵完备的智能合约编程语言 [41]，使以太坊成为首个支持智能合约的区块链平台。

智能合约是一套以数字形式定义的承诺，包括合约参与方可以在上面执行这些承诺的协议 [6]。与纸质合约不同，智能合约以代码的形式存在。智能合约中的条件会以代码的形式写明，一旦达成条件，合约会自动执行，这有效地降低了违约的风险 [42]。智能合约的意义在于，所有节点对该智能合约中的状态达成共识，其他人无法通过任何方式进行篡改。智能合约最先没有广泛使用，因为其执行环境不可信，有被认为篡改的可能。区块链的出现为智能合约提供了一个可信的平台，其不可篡改的特性为智能合约提供了一个完美的运行平台，智能合约的条件、内容等得到良好的保护，合约中的各方必须按照合约条件履行各自的义务。智能合约与区块链大大扩充了区块链的应用场景。

智能合约通过交易实现与合约的交互。每个智能合约部署在区块链中之后都有一个固定的合约地址，交易发起者可以通过智能合约地址指定需要发送交易的合约。智能合约的调用类似于函数调用，可以在交易请求中指定调用智能合约的函数与参数实现交易。交易完成后将被矿工打包进区块，在包含交易的区块被打包并确认后，合约中的值和状态将发生相应的改变。

2.1.3　以太坊

2013 年底，以太坊创始人 Vitalik Buterin 发布了以太坊初版白皮书 [7]，并与一批认可以太坊理念的开发者一起启动了项目。在不断的开发迭代的过程中，以太坊已经成为一个功能强大并成熟的区块链平台。截止 2020 年 1 月，以太坊市值已超过 1000 亿人民币 [30]。目前，以太坊仅次于比特币成为第二大市值的区块链平台。相较于比特币，以太坊添加了对智能合约的支持，并支持图灵完备的智能合约编程语言。随着以太坊的发展，以太坊对私有链的支持也逐渐成熟，越来越多的人采用以太坊搭建自己的私有区块链。

(1) Gas 机制

以太坊中，用户发送的请求称为交易。Gas 机制是以太坊的核心机制，用于衡量每一笔交易消耗的计算量。每笔交易会收取一定数额的手续费，数额大

小与计算量直接相关。在以太坊中，交易通过以太坊虚拟机（Ethereum Virtual Engine）执行。智能合约被编译为 EVM 字节码，执行每个字节码会消耗一定数量的 Gas。交易完成后，EVM 将计算出此交易消耗的总 Gas 数量。用户会为交易设置 Gas Limit，代表用户为这笔交易提供的 Gas 数量。每笔交易都会设定 Gas Price，用于计算交易手续费，手续费将从交易发送方的帐户余额中扣除，如过用户提供的 Gas 数量不足则交易会被回滚，并且手续费不会退还 [41]。每个区块也会设置区块 Gas Limit，其打包的所有交易消耗的 Gas 数量之和不能超过区块 Gas Limit。通过收取手续费，可以防止恶意用户发送无效请求对以太坊进行攻击。如果用户发送恶意请求试图占用矿工计算资源，其交易不仅不会生效，交易费用也会正常收取，增加了作恶成本。由此 Gas 机制可以有效的防止恶意交易对以太坊造成影响。同时，消耗 Gas 产生的交易手续费最终会支付给矿工，这也激励了更多矿工参与挖矿。

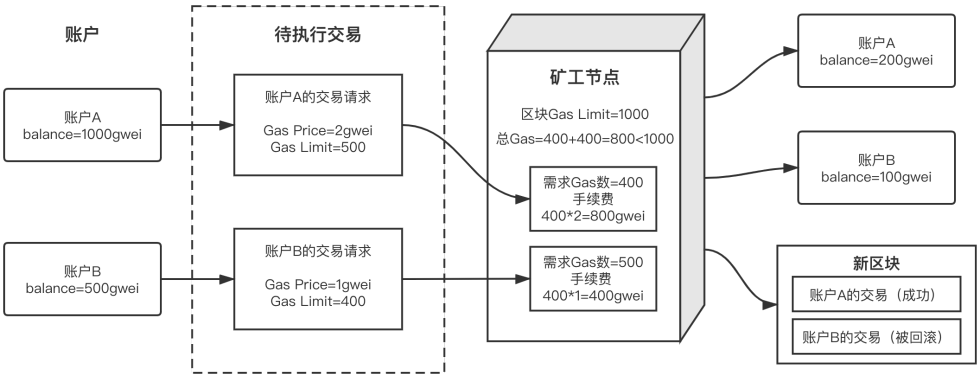


图 2.3: Gas 机制工作示例

以图2.3为例进一步说明 Gas 机制：账户 A 与 B 分别有一定数目的余额，并分别向以太坊发送了一笔交易，交易均设置了 Gas Price 与 Gas Limit。假设矿工节点只打包这两笔交易，且打包到同一个区块中。在矿工的打包过程中，执行 A 的交易请求所需的 Gas 数量小于 A 设置的 Gas Limit，剩余的 Gas 被返还给账户，交易成功执行；B 的交易请求中设置的 Gas Limit 无法支撑这笔交易执行完成，导致 Gas 耗尽，交易被回滚。最终，这两笔交易实际消耗的 Gas 数量总和不超过区块 Gas Limit。

(2) 以太坊智能合约

以太坊是第一个基于区块链和智能合约技术的区块链平台，以太坊上的智能合约可以由 Solidity 语言编写。Solidity 是一种图灵完备的语言，语法类似于 JavaScript，可以实现功能强大的智能合约 [43]。Solidity 编写的智能合约可以编

译成 EVM 字节码在 EVM 上运行。为了预防恶意攻击，以太坊虚拟机执行合约的过程直接与金钱挂钩，也就是 Gas 机制。Solidity 中可以设置全局变量，授权的用户可以通过交易改变其中的值。相比于其他编程语言，Solidity 中包含一些特有的关键字与数据结构，并设有事件机制。

```
1.  pragma solidity ^0.4.2;
2.
3.  contract BadPerformance {
4.
5.      mapping(string=>uint) map0;
6.      mapping(string=>uint) map1;
7.
8.      function badPerfLevel(uint level, string arg0, uint arg1) public {
9.          for (uint p = 0; p < 10**level; p++) {
10.             map0[arg0] = arg1 + p;
11.             map1[arg0] = map0[arg0] + p;
12.             map1[arg0] = map0[arg0] * map1[arg0];
13.             map0[arg0] = map1[arg0] / map0[arg0];
14.         }
15.     }
16. }
```

图 2.4: Solidity 语言代码示例

如图2.4为一个 Solidity 实现的简单合约代码示例。第 1 行代表了 Solidity 语言的版本。第 3 行的 contract 关键字代表一个合约实例。第 5、6 行声明的 mapping 类型的变量为合约中哈希表类型的状态变量。状态变量会占据以太坊的存储空间，用户可以通过交易请求改变它的值。第 8-15 行为一个合约函数，public 关键字代表其可以通过发送交易请求进行调用。

(3) 性能影响因素

以太坊私有链因为去中心化的特性常被用在需要高可信的场景下，但是去中心化也带来了性能问题。以太坊实现了 PoW 共识协议，需要大量的计算资源。以太坊矿工执行名为 Ethash 的哈希函数挖掘与验证区块，可以通过设置 Difficulty 字段控制挖矿难度。挖矿难度直接对应计算出的哈希值需要满足的条件，难度越高，需要进行尝试的次数越多，也就需要消耗更多计算资源，出块的速度也越慢。以太坊的 Gas 机制用于防止部分恶意交易利用智能合约的代码漏洞攻击区块链，但其中的区块 Gas Limit 也限制了区块可以容纳的交易数量。由于共识协议与 Gas 机制，以太坊的交易处理能力受到限制，从而导致性能瓶颈[44]，其交易吞吐量和延迟无法到达生产级别。

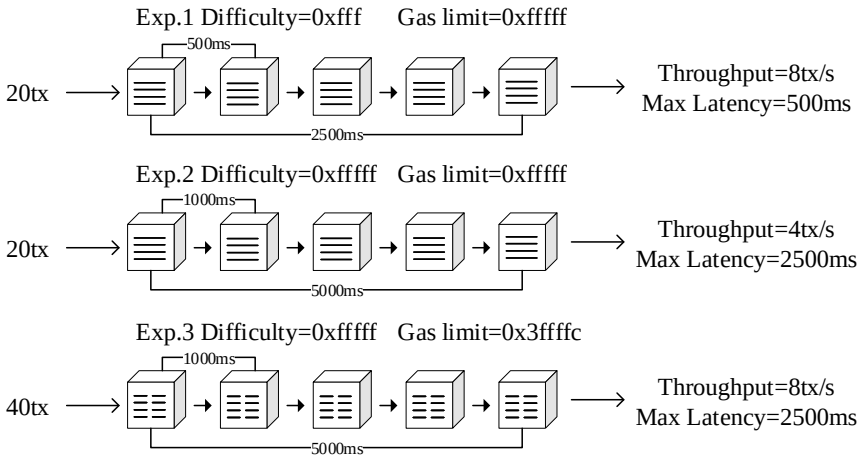


图 2.5: Difficulty 与 Gas Limit 对性能的影响

如图2.5所示为三个不同 Difficulty 与 Gas Limit 设置的以太坊私有链的挖矿示例，每个私有链中的矿工具有相同的算力。区块间的连线代表这生成两个区块之间的时间间隔，区块中的线段数量代表了区块容纳的交易数量。Difficulty 影响工作量证明中解决谜题的难度，在矿工算力相同的情况下，随着 Difficulty 的增大，每两个区块的出块间隔增大，导致交易延迟上升。而以太坊的 Gas 机制中区块 Gas Limit 字段则限制了一个区块所能包含的交易数量。图中同样产生 5 个区块，Gas Limit 更高的区块链，容纳的交易数量更多，因此吞吐量更高。

2.2　性能测试

随着软件系统的复杂性、用户规模的提升，开发人员不仅需要关注系统实现的功能是否完整准确，也要关注系统的性能能否达到要求。软件的性能是一种非功能特性，它并不表现在系统能否正确完成其功能，而是完成功能的效率。系统性能的重要性不亚于功能，因为其直接关乎用户体验。性能测试技术是一种通过模拟系统负载从而获取系统性能指标的测试技术。性能测试的目的在于获取系统在不同负载水平下的性能表现，从而判断系统是否能够达到所需的性能要求，使用户对系统能承受的负载范围有足够的预期 [15]。

性能测试也有助于系统的开发人员及时发现与修复系统潜在的性能问题。造成性能缺陷的原因有很多种，如代码设计缺陷、不必要的线程同步操作或者不合理的负载均衡算法等 [45]。由于这些缺陷可能要一定并发数量，或是系统处理的请求数量达到一定程度时才会触发，单一的功能测试很难暴露出这类性能问题。性能测试技术则专门用于测试这类潜在的性能问题。

2.2.1　性能测试方法

根据用户需求不同，性能测试主要分为基准测试、负载测试与压力测试等三类，对应软件开发的不同时期和软件的不同使用场景 [46]。

基准测试用于软件系统发生较大变更或新增、删除模块时，判断变更对整个软件系统的性能影响。基准测试以变更前后的性能指标波动作为判断测试是否通过的标准。依据基准测试方法，需要以变更前后的系统至少各做一次测试。记录变更前的指标作为变更后的基准，如变更后测试达到基准则测试通过。

负载测试用于考察系统在特定的负载下能否达到预先设定的性能指标要求。负载测试会为测试结果设定一个指标基准，通过测试后性能指标是否达到该基准评价系统性能是否合乎要求。负载测试根据系统在需求分析时的性能需求作为测试是否成功的基准。如果达到了性能需求中预计的指标为测试通过。

压力测试则是在系统开发完成到上线之前，对系统在高请求压力下的表现进行测试。压力测试通常没有是否通过测试之分，其目标在于探究系统真正能够承受的请求压力大小。在测试过程中，在不断对系统施加压力的情况下，如果系统处理回话的错误率不超过一定标准，则认为系统能够承受当前压力。当压力超过某一临界值后系统处理请求错误率超过标准，则认为系统达到了其最大压力承受能力，系统的维护者要尽可能避免系统承受的压力超过该临界值。

2.2.2　性能测试指标

性能测试指标是量化系统性能的标准，不同指标可以从不同角度对待测系统的性能进行准确评估。在对区块链系统的性能评估中，性能指标根据其评估角度分为宏观指标与微观指标。宏观指标表示从整体区块链的角度评估性能，包括吞吐量、交易延迟等。吞吐量为系统同时能处理的交易数量，直观展示系统性能。交易延迟为系统在单位时间内返回的交易的平均处理时间，体现了系统对于单个交易的处理能力。微观指标表示从区块链每个节点自身角度进行性能评估，包括区块链节点的 CPU、内存、网络、I/O 资源的利用率等指标 [20]。

$$thr = \frac{n_t}{t} \tag{2.1}$$

在区块链的性能测试中，开发者通常使用吞吐量、交易延迟等指标来评判一个系统整体的性能 [20]。其中，吞吐量表示单位时间内系统能够处理的事务数量。如公式2.1所示，thr 为吞吐量（throughput），n_t 表示在时间段 t 内系统返回

的交易数量。吞吐量不同于请求速率，吞吐量的大小直接对应了系统处理事务的能力，较低的吞吐量可能无法满足大量用户同时使用。

$$lan = \frac{\sum_{i=1}^{n} lan_i}{n} \tag{2.2}$$

延迟表示了每一个事务从发送到成功返回结果的平均时间间隔。如公式2.2所示，lan 为交易延迟（latency），lan_i 表示第 i 个交易的延迟时间，n 表示一共 n 个交易。高延迟可能会给用户带来较差的体验。

2.2.3　Blockbench 与 Hyperledger Caliper

区块链作为一种分布式系统 [2]，其性能也值得关注，然而目前对区块链领域的性能测试技术的研究还比较少。目前，在区块链性能测试领域中，处于领先地位的有 Dinh 等人 Blockbench[20] 与华为公司的 Caliper[22]。

Blockbench 是最早出现的区块链性能测试工具，可以对以太坊的 Geth、Parity 客户端和超级账本进行性能测试。该工具实现了固定请求速率的负载测试，并可以根据不同区块链类型进行横向扩展。该工具根据区块链的层级划分，使用不同种类型的智能合约作为工作负载，用以测试区块链对应层级的性能表现。该工具可以获取系统在测试过程中的平均吞吐量与交易延迟，对区块链性能进行整体评估。

目前，华为公司也发布了 Hyperledger Caliper。该工具用于超级账本系列的区块链系统，如 HyperLedger Fabric 等，并被超级账本项目收录为官方性能测试工具。该工具可以根据 Docker 配置自主搭建测试链，支持吞吐量、交易延迟等宏观性能指标，也支持 CPU、内存等微观性能指标。

2.3　故障注入

故障注入是一种常见的测试技术，有助于帮助开发者理解系统在异常环境下或发生故障时的行为。故障注入技术通过在真实的系统中人为注入故障来模拟真实场景，在此基础上进行测试并获得结果 [26]。因此，与统计历史数据的方法相比，故障注入测试的结果可能更接近线上运行情况 [47]。故障注入技术最早可追溯到 20 世纪 70 年代，该技术最初用于以物理手段模拟硬件故障 [48]。在软件测试领域，故障注入技术通过向在测试环境中部署的软件系统注入预先设定的故障，来模拟真实线上环境下可能发生的故障 [24]。随着分布式系统的发展，系统复杂程度提升，系统缺陷更难以发现，故障带来的影响也更大 [12]。因此，在分布式系统中，该技术应用更为广泛。

2.3.1　分布式系统中的故障注入

最初，软件多为集中式系统，其缺陷很容易暴露并修复。而分布式技术不断发展，规模呈指数级上升，系统之间复杂的依赖关系与高频通信使故障发生的概率也随之不断升高 [49]。为了在大规模的分布式系统中维持稳定性，故障注入技术的重要性逐渐体现。在分布式系统中，一个微小的变更可能对自身或与之相关的子系统产生不可预知的影响，任何一个子系统的故障都可能导致整个系统故障 [25]。而且，这种故障很难在测试中发现，因为测试环境与线上环境有很大差异，包括请求量、部署规模、网络环境等。实证研究表明：1）大部分的故障事件发生是由不多于 3 个小规模故障导致；2）特定的事件顺序对的故障产生有直接影响；3）绝大多数的故障事件是不超过 3 个分布式节点的故障导致的；4）导致故障的事件组合可能重新导致该故障。由此可见，分布式系统故障往往是在小范围内发生后逐步扩大影响，并且这种小范围故障是可预见且可控的 [27]。通过故障注入的方式，可以发现故障的触发条件，提前预防系统大规模故障，减少损失。为此，为了尽可能的模拟真实环境，开发人员引入故障注入技术，在测试环境尽可能复现线上环境可能的故障 [47]。由此，故障注入技术被广泛运用在分布式系统测试领域。

单一的故障注入模式难以满足大规模分布式系统的测试需求。2015 年 Netflix 的软件工程师提出了“混沌工程”概念 [14]，将故障注入技术工程化。混沌工程的主要思想是，首先确定系统的稳定状态的假设，然后在真实的线上系统中进行自动化且随机的故障注入，并保证系统在可控的“爆炸半径”内，以此来理解该系统的稳定性与性能。混沌工程主要包含五个原则：（1）建立稳定状态的假设，确定系统的预期行为；（2）多样化显示世界的实验，尽可能涵盖更多的故障场景；（3）在生产环境运行实验，因为生产环境最能反映系统的运行状态；（4）持续自动化运行实验，因为故障场景的多样性使手动执行测试变得异常艰难；（5）最小化“爆炸半径”，将注入的故障对生产环境降到最低，尽可能降低对正常用户的影响 [14]。如表2.1为混沌工程概念中针对故障的成因、影响范围等因素，对故障的分类。表中展示了分布式系统中，应用、资源、网络、系统四个类型的故障的名称与具体表现。

2.3.2　区块链中的故障注入

由于区块链也可以视为一种分布式系统，故障注入技术也可以运在区块链中。但由于区块链具有去中心化与不可篡改等特性，使得区块链中的故障注入与分布式系统具有一定差异。在区块链系统中，混沌工程的这五个原则并不完全匹配：（1）对于第三条原则，由于区块链上的数据会永久保留，测试时可能在

表 2.1: 分布式系统故障分类与描述

序号	故障类型	故障名称	故障描述
1	应用	流量突增	模拟系统可能出现的流量激增或随机抖动
		重试风暴	模拟系统短暂拒绝服务时，用户反复尝试造成重试风暴
2	资源	CPU 耗尽	模拟服务器可能出现的 CPU 耗尽现象
		内存耗尽	模拟服务器可能出现的内存耗尽现象
		磁盘耗尽	模拟服务器可能出现的磁盘耗尽现象
3	网络	延迟	模拟系统网络出现延迟增长的现象
		丢包	模拟系统网络出现大面积丢包的现象
4	系统	下游故障	模拟系统下游服务出现故障
		服务宕机	模拟系统中单独或部分服务器宕机的现象

生产环境留下脏数据，因此不宜在生产环境试验；（2）由于不满足第三条原则在生产环境试验，为了模拟真实环境，可以考虑通过在新搭建的区块链中进行性能测试的方式，模拟出一个近似于生产环境负载的测试环境；（3）第五条原则中，对于“爆炸半径”可以不做控制，不必考虑对用户的影响。由此可以认为，虽然区块链系统与传统分布式软件系统有一定的不同，故障注入技术可以有效运用在区块链系统的性能测试实验中。

混沌工程思想中提出了四种针对分布式系统的故障类型，其中提出的部分故障类型在区块链系统中也会出现，如流量突增故障。2017 年以太坊层发生“加密猫”事件，一个名为 CryptoKitties 的分布式应用几乎使整个以太坊瘫痪。由于同时向该分布式应用发送交易的用户数量突增，近 20000 个交易被阻塞 [50]。

也有部分故障类型虽然也可能在区块链系统中产生，但是故障造成的影响并不相同，如服务宕机。传统的分布式系统中，每一个服务可能有多个服务器进行热备，节点之间可以互相替换，并可以在 Master 节点宕机时使用选举协议重新选出 Master 节点。区块链系统的节点之间相对独立，对于需要共识的区块链节点而言，节点的宕机与重启均涉及到共识网络的退出与重新识别，其影响大于传统分布式系统的服务宕机。

由于区块链系统引入了对智能合约的支持，本文又提出了智能合约故障。智能合约的代码可以由用户编写并部署到区块链上，在执行交易时，会在以太坊节点中运行该代码，其代码的质量会影响节点运行的效率。而且，由于所有节点都会验证区块中的每个交易，该代码会在所有节点执行一次，性能影响会进一步放大。

根据上述分析，总结出区块链故障类型与描述如表2.2所示。其中应用与网络故障类型是从分布式系统中迁移到区块链系统，共识与智能合约类型根据区

表 2.2: 区块链故障分类与描述

序号	故障类型	故障名称	故障描述
1	应用	流量突增	模拟多节点可能出现的流量激增或随机抖动
2	共识	节点宕机	模拟节点可能出现宕机并退出共识网络
3	网络	延迟	模拟区块链在网络环境出现延迟增长的现象
		丢包	模拟区块链在网络环境出现大面积丢包的现象
4	智能合约	低性能合约	模拟区块链上部署的低性能智能合约

块链特性总结。除共识故障外故障，其余三种故障可以根据严重程度配置分故障参数：（1）应用故障的流量增长倍率大小；（2）网络故障的丢包率大小；（3）智能合约根据执行合约交易时的无效循环次数。

2.4　本章小结

本章主要描述了本性能测试系统使用到的主要概念与技术，分别从性能测试与故障注入两个角度，深入介绍其概念、发展以及现有工具。在现有技术的基础上，结合当前流行的区块链技术，分析这些技术与区块链的内在关系。从现有技术中拓展出与本系统相关的技术领域，并进行深入剖析，为得出可运用与本系统的设计与实现中的关键概念与技术奠定了基础。

第三章　需求分析与概要设计

根据第二章对以太坊私有链的性能分析与区块链中的故障总结，本章对以太坊私有链性能测试系统进行需求分析与概要设计。首先需要对系统进行涉众与用例进行分析得出功能与非功能需求。根据需求分析的结果，对系统进行概要设计，给出系统的整体架构、4+1 视图与模块设计。

3.1　需求分析

3.1.1　涉众分析

以太坊是目前较为成熟的私有链平台，企业或个人可以使用以太坊客户端搭建私有链。私有链具有一定的用户规模，用户会期望发送到链上的交易能及时响应，因此以太坊私有链性能会影响私有链使用者的用户体验。以太坊是开源软件，任何人均可参与开发。性能问题作为以太坊当前面临的关键问题，除以太坊开发团队外，有众多开源社区的开发者致力于性能优化。性能问题也同样是以太坊维护人员关注的问题。维护人员致力于系统的稳定，期望系统能在常规工作负载下正常运转。根据上述分析，本节将系统涉众分为三类：以太坊使用者、以太坊开发者、以太坊维护者，并总结出涉众的特征与期望如表3.1所示。

表 3.1: 涉众的特征与期望

涉众类型	特征	期望
以太坊使用者	使用以太坊上的分布式应用	发送的交易能够及时响应、准确打包
以太坊开发者	开发以太坊源码 充分理解以太坊原理	直观衡量以太坊性能的优化程度
以太坊维护者	维护以太坊私有链 了解以太坊部署方法	测试以太坊性能能否满足用户需求

以太坊使用者。以以太坊私有链为基础的分布式应用的用户，对以太坊与私有链具有一定的了解。此类用户希望发送的交易能及时响应并正确打包，能够查看性能测试结果了解私有链性能是否能够达到要求。

以太坊开发者。以太坊私有链客户端代码开发人员，对以太坊私有链充分了解，不断更新以太坊代码以提升其性能。在本平台进行性能与故障注入测试，希望以此充分分析更新后的以太坊客户端的性能与稳定性表现。

以太坊维护者。以太坊私有链搭建者与维护者，对以太坊私有链有较好的理解，对所维护的以太坊私有链有一定的性能需求，希望能够扩大用户规模，并为用户提供更方便快捷的体验，并通过不断优化与测试找出提升性能与稳定性的最佳方案。

3.1.2　用例分析

根据对涉众的特征和期望进行分析，本节将用例划分为测试任务与测试结果相关的两部分，得出系统用例如图3.1所示。

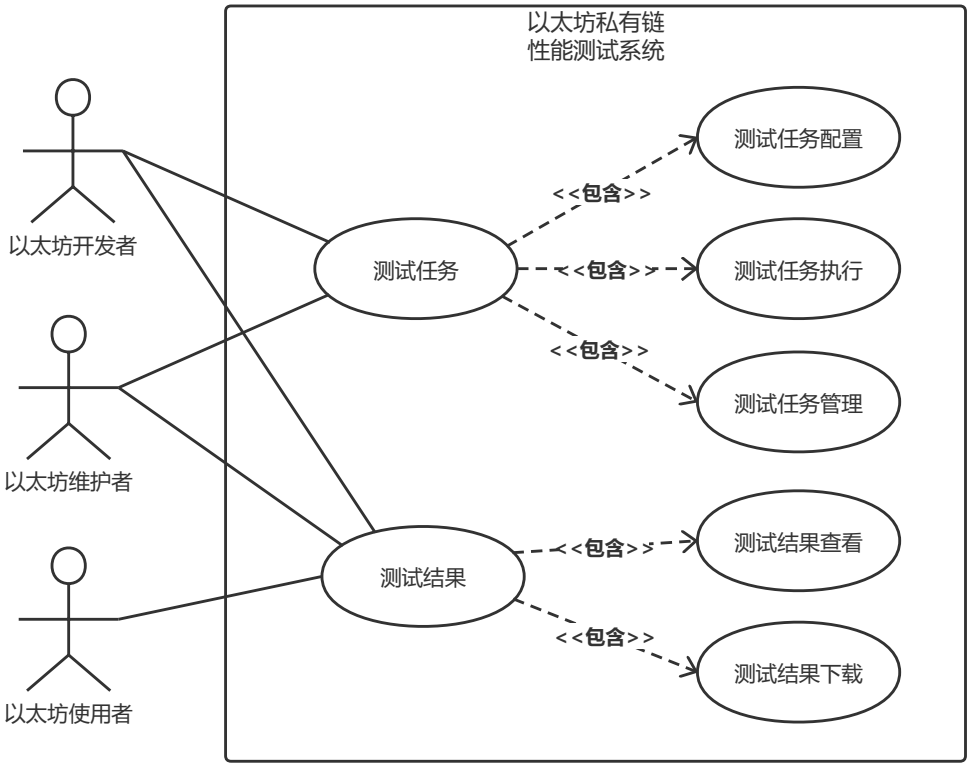


图 3.1: 系统用例图

在整个性能测试的过程中，首先，以太坊开发者和维护者可以对测试任务进行配置，填写相关的工作负载与故障注入信息。然后，根据配置信息可以开始执行性能测试与故障注入任务，系统会在后端进行区块链搭建、请求发送、故障注入与性能指标收集等操作。最后，整个测试任务完成后，测试结果将存储在数据库中。测试结果以图表方式在线展示，也支持用户下载保存。以太坊开发者可以对测试结果进行管理，对无效测试配置与异常结果进行删除。结合系统用例图以及上述对用例的分析，下面给出详细用例描述。

表 3.2: 测试任务配置

ID	UC1
名称	测试任务配置
参与者	以太坊维护者、以太坊开发者
触发条件	以太坊维护者、以太坊开发者点击创建测试任务
前置条件	以太坊维护者、以太坊开发者已授权使用本系统
后置条件	无
优先级	高
正常流程	1. 用户点击创建新测试任务按钮。 2. 系统展示性能测试任务配置界面。 3. 用户填写测试所需配置并确认。 4. 系统验证配置并展示故障注入配置界面。 5. 用户填写故障配置并确认。 6. 系统验证配置有效性并存储配置。
扩展流程	3a. 用户填写的测试配置有误。 1. 显示错误并提示重新填写。

表3.2给出了测试任务配置的用例描述。测试任务配置主要分为、区块链配置、性能测试配置与故障注入配置三部分。区块链配置为启动以太坊测试链的参数，包括 Diffculty、Gas Limit 等影响以太坊私有链性能的配置项。性能测试配置包含请求速率、工作负载、测试持续时间等。故障注入配置包含每个故障的类型、参数、注入时间、持续时间等, 生成的配置将会保存在数据库中。部分配置项会有一定的条件限制，如 Difficulty 与 Gas Limit 不能小于 0. 因此需要在输入配置时进行校验，防止测试过程中出现问题导致测试失败。

表 3.3: 测试任务执行

ID	UC2
名称	测试任务执行
参与者	以太坊维护者、以太坊开发者
触发条件	以太坊维护者、以太坊开发者点击执行测试任务
前置条件	以太坊维护者、以太坊开发者已授权使用本系统，并已有合法的测试任务配置或以及输入本次测试配置
后置条件	无
优先级	高
正常流程	1. 用户点击历史测试结果列表。 2. 系统展示历史测试结果列表。 3. 用户点击某个测试任务详细信息。 4. 系统展示本测试任务的测试配置。 5. 用户点击执行测试任务按钮。 6. 系统开始执行测试任务并展示任务正在执行中。
扩展流程	3a. 用户不使用已有测试配置进行测试。 1. 用户执行 UC1 中的正常流程，并点击执行测试任务。

表3.3给出了测试任务执行的用例描述。在执行测试任务之前，首先需要选定已有的测试配置，或输入新的测试配置。测试任务启动后，系统会按照区块链配置搭建起一个新的以太坊测试链，并对其进行初始化操作，如智能合约、故障注入的初始化。搭建完成后，系统会解析性能测试配置，并根据配置信息向测试链发送交易请求。测试过程中，根据故障注入配置完成故障注入。整个过程全部自动化完成，无需用户介入。完成测试后系统收集相应的性能指标，存储到数据库中。由于整个测试时间较长，测试时会展示一个执行中状态，测试完成后，测试结果将自动存入系统中，任务状态将变更为已完成。

表 3.4: 测试结果查看

ID	UC3
名称	测试结果查看
参与者	以太坊维护者、以太坊开发者、以太坊使用者
触发条件	以太坊维护者、以太坊开发者、以太坊使用者点击某一测试任务
前置条件	以太坊维护者、以太坊开发者、以太坊使用者已授权使用本系统，并已有成功执行的性能测试任务
后置条件	无
优先级	高
正常流程	1. 用户点击历史测试结果列表。 2. 系统展示历史测试结果列表。 3. 用户点击某个测试任务详细信息。 4. 系统展示本测试任务的测试配置与结果。
扩展流程	3a. 用户点击未执行完成的测试任务详细信息。 1. 系统只展示配置信息。 4a. 用户点击查看原始测试数据。 1. 系统展示测试原始数据。

表3.4给出了测试结果查看的用例描述。在历史测试任务结果列表中会列出当前所有任务。用户点击测试任务可以进一步查看其测试配置与结果。测试结果查看需要等到待查看的测试任务成功完成后才可进行，未完成或测试任务异常结束的都不会生成相应结果。测试结果以整体指标数据结合图表的方式展示。整体指标数据以表格的形式展示，包括测试过程中的吞吐量、交易延迟的最大值、最小值、平均值等。由于系统引入了故障注入机制，故障注入前、故障中与故障恢复后的指标也展示出系统性能与稳定性，因此也将以表格形式展示。图表数据为秒级指标的折线图，可以更好的展示出指标随时间的变化趋势。由于区块链的挖矿机制导致区块生成速度随机性强，性能指标会受较大影响出现大幅度抖动，因此默认展示经过平滑处理的测试结果，以便分析性能以及故障的影响。平滑程度可以根据总数据量大小自动进行调节。然而平滑处理可能会对峰值、谷值产生影响，因此保留了原始数据图表，用户可以手动切换。

表 3.5: 测试结果下载

ID	UC4
名称	测试结果下载
参与者	以太坊维护者、以太坊开发者、以太坊使用者
触发条件	以太坊维护者、以太坊开发者、以太坊使用者点击创建测试任务
前置条件	以太坊维护者、以太坊开发者、以太坊使用者已授权使用本系统，并已有成功执行的性能测试任务
后置条件	无
优先级	中
正常流程	1. 用户点击历史测试结果列表。 2. 系统展示历史测试结果列表。 3. 用户点击某个测试任务详细信息。 4. 系统展示本测试任务的测试配置与结果。 5. 用户点击下载测试结果按钮。 6. 系统开始下载测试结果到本地。
扩展流程	无

表3.5给出了测试结果下载的用例描述。测试任务成功完成后即可下载测试结果。以太坊开发者、维护者与使用者均可以下载测试结果。测试结果可以以图片或原始数据形式下载。图片形式与页面展示的结果保持一致，数据形式则以 JSON 格式下载，以便于对数据的进一步分析处理。

表 3.6: 测试任务管理

ID	UC5
名称	测试任务管理
参与者	以太坊维护者、以太坊开发者
触发条件	以太坊维护者、以太坊开发者点击删除某一测试任务
前置条件	以太坊维护者、以太坊开发者已授权使用本系统，并已有测试任务
后置条件	无
优先级	低
正常流程	1. 用户点击历史测试结果列表。 2. 系统展示历史测试结果列表。 3. 用户点击删除某个测试任务。 4. 系统展示是否确认删除。 5. 用户点击确认删除。 6. 系统从数据库删除任务并刷新页面。
扩展流程	5a. 用户取消删除。 1. 系统返回原页面。

表3.6给出了测试任务管理的用例描述。测试任务管理由创建该测试任务的用户进行。由于以太坊私有链技术尚处在研发阶段，从搭建区块链到最终生成测试结果的过程容易出现问题导致测试任务异常终止。正常执行的测试任务中，

也可能由于服务器资源被其他进程占用或一些其他因素，导致一些测试结果不具备参考价值。所以测试执行者可以对测试任务进行管理，删除一些无效测试配置与异常结果。

3.1.3　功能需求

根据系统涉众分析和用例分析，将以太坊私有链性能测试系统的功能需求分为测试配置生成、测试任务执行、测试结果展示和测试任务管理四部分。

测试配置生成。以太坊开发者与维护者根据自身需求生成自己的测试配置。测试配置是测试任务的基础，主要包括性能测试配置与故障注入配置。用户生成的配置将被持久化以便后续执行与展示。

测试任务执行。以太坊开发者与维护者根据测试配置执行测试任务。测试任务执行是本系统的核心功能，主要包括测试链搭建，性能测试、故障注入三部分。用户开始执行测试任务后系统开始在后端执行，测试完成后将输出性能指标数据。

测试结果展示。所有用户查看已成功执行的测试任务结果。测试任务成功执行后输出的原始性能指标数据不够直观，并且由于区块链的特性存在较大的抖动。因此，本功能以图表的形式展示性能指标并将数据进行一定程度的平滑处理以便更清晰的展示测试结果。

测试任务管理。以太坊开发者与维护者对其执行的测试任务进行管理。本功能需求要建立在账户管理的基础上。对应账户有对相应测试任务的权限，进而执行管理操作。

3.1.4　非功能需求

以太坊私有链性能测试系统的核心目的是为以太坊开发者、维护者与使用者提供一个方便快捷的方式了解、测试、分析以太坊私有链的性能与稳定性。首先，由于测试配置较为复杂，需要提供充分的说明降低使用难度。其次，由于搭建以太坊测试链较为耗时，且测试任务时间较长，需要保证每次任务的成功率。再次，以太坊客户端可能不断迭代更新，需要保证系统的可维护性。最后，工作负载、故障类型、以太坊客户端可能随着以太坊、性能测试、故障注入技术的发展推陈出新，需要保证系统的可扩展性。根据上述分析，总结出四种非功能需求，即易用性、可用性、可维护性、可扩展性。

易用性。由于本系统新建测试任务包含性能测试与故障注入两种配置，配置项多且复杂。因此，需要提供充分说明介绍配置项的内容。同时，由于整个流程的专业性较强，需要对整个操作流程进行优化提升用户体验。

可用性。由于整个测试链搭建、测试任务执行周期长，测试失败需要重新开始测试，使系统难以保证测试时的效率。因此，需要保证系统的可用性，降低测试失败的概率。

可维护性。以太坊客户端仍然处于不断更新迭代的阶段，可能会出现接口增加删除或修改，或是内部机制的变更。因此，需要保证系统的可维护性，从而保障系统持续可用。

可扩展性。随着对区块链及以太坊的研究不断深入，测试负载使用的智能合约、注入故障的类型、以太坊客户端的类型等可能变化或增加。为了适应变更的需求，需要保证良好的可扩展性。

3.2　概要设计

以上述需求分析中总结出的功能需求和非能性需求为基础，本节进行了系统概要设计。首先，从宏观角度给出系统的架构设计，整体把握系统的层次。其次，给出从系统的逻辑视图、进程视图、开发视图、物理视图，从多个角度对系统的设计进行描述。由于本系统的功能需求在实现上有较大的差异，因此将系统拆分为两个模块分别进行设计，分别采用最适合的方式对两个模块进行实现。

3.2.1　系统架构

图 3.2: 系统整体框架

本系统的整体架构设计如图3.2所示。系统整体分成四部分：前端展示主要负责展示数据以及与用户的交互；用户管理负责用户的登录、注册与权限管理；测试管理模块则将用户输入转换成配置文件以及将测试结果进行处理转换为可

以展示的形式，以及一些持久化的操作；测试执行模块则是系统的核心，负责执行测试任务。前端展示通过 H5 页面与用户交互并以图表的形式展示数据，采用较为成熟的 Echarts.js 作为渲染图表的引擎。服务端采用 Python 语言支持 Django 框架，便于测试结果数据处理集成。对于测试管理模块，选用最适合处理数据的 Python 语言，通过 NumPy 等扩展包可以快速处理大量的交易数据并转化为可以展示的数据格式。配置文件和测试结果数据都以 JSON 的格式进行存储，以便于字段扩展。持久化采用 NoSQL 数据库 MongoDB，对 JSON 格式的数据支持较好，存取速度快，便于快速的数据存取。测试执行模块则是整个系统的核心模块，采用对异步操作支持较好的 NodeJS 编写。通过 Docker 启动以太坊区块链，Web3.js 与以太坊节点进行交互。故障注入操作则使用 Docker API 以及支持 Docker 故障注入的 Pumba 混沌工程工具执行。

3.2.2　架构视图

本节采用 Philippe Kruchten 的“4+1”视图 [51] 对本系统的概要设计进行进一步描述。分别从逻辑视角、开发视角、进程视角以及物理部署视角给出系统的架构设计。

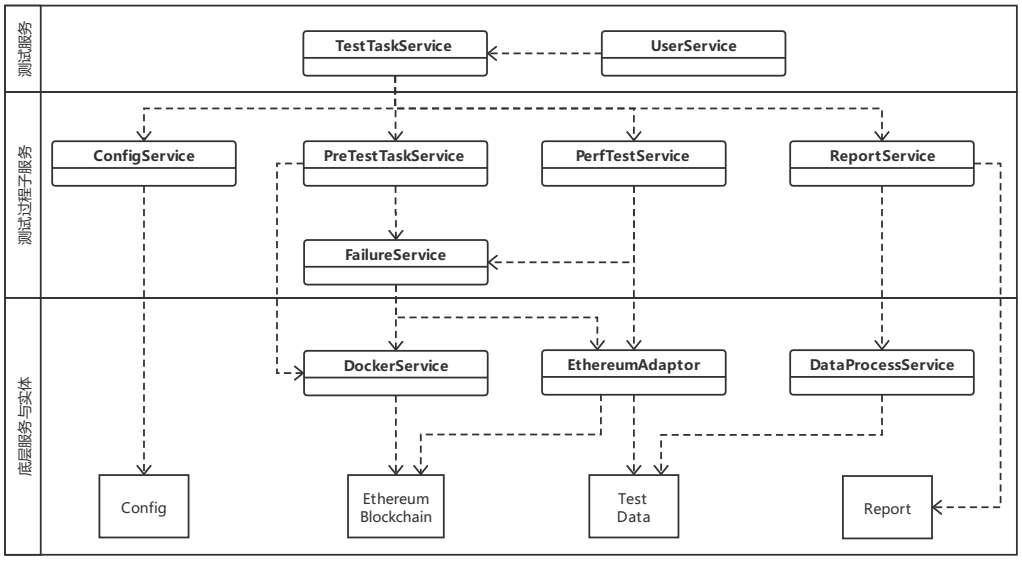


图 3.3: 逻辑视图

逻辑视图。图3.3展示了以太坊私有链性能测试系统的逻辑视图。逻辑视图通过将系统之间的调用逻辑抽象成服务对系统进行描述。该视图揭示了系统的核心逻辑，往往以面向对象的方法为基础，将系统抽象为对象或类进行描述，因此以类图的方式描述逻辑视图。

在图3.3中，整体的系统逻辑大致分为三个层次：第一层次包含测试服务，测试服务依赖下一层的所有子服务，通过调用子服务实现整个测试任务，并将结果交还用户服务，再由用户决定下一步操作。第二层次包括直接由测试服务调用的子服务，这些子服务负责测试过程的每个阶段，包括测试准备、性能测试、故障注入、测试报告等服务，这些服务将整个测试任务大致分为四个步骤，每个服务之间没有直接关系。第三个层次包含其余所有底层服务以及实体，这一层的服务之间为了尽可能的复用，与上层服务还有相互之间调用逻辑较为复杂，因此单独抽出这一层，尽可能将复杂的逻辑调用向高层次屏蔽。这一层的，实体包括测试配置、以太坊区块链、测试数据以及测试报告，与实体相关的服务主要包括与以太坊私有链交互、故障注入、测试数据处理与测试报告生成服务。

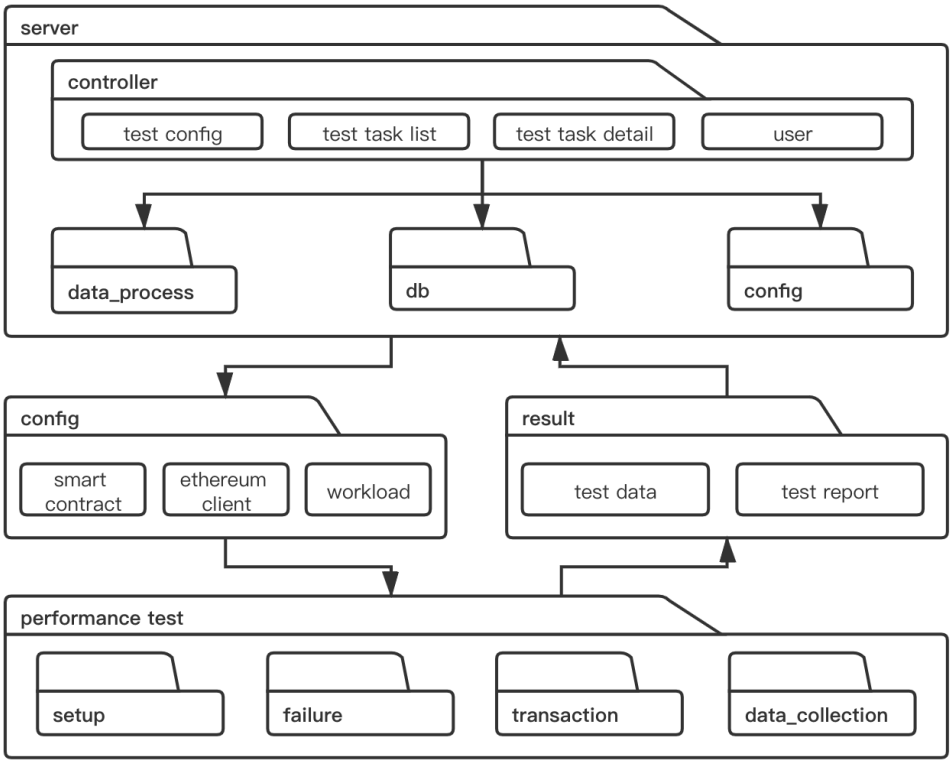


图 3.4: 开发视图

开发视图。本系统开发视图如图3.4所示。开发视图主要展示了系统的模块组成与开发过程中的软件开发包的层级结构。这种视图将相关性强的业务逻辑整合到同一个包中，有助于开发人员提示效率。

在图3.4中，将开发视图大致分为三个层级。最上层为服务器相关代码逻辑，由于采用了 Django 框架，整个开发包结构类似 MVC 风格。除此之外，本层次

还包括对测试配置的整合与测试结果数据的软件包。Controller 调用下面三个包完成测试配置生成、测试结果数据处理与存储。最下层为性能测试任务执行代码逻辑，包含从启动测试链到最终收集结果数据四个主要步骤的软件包。这四个包各自完成本步骤的职责，互相之间不存在依赖，它们共同依赖中间层提供的数据。中间层为系统智能合约、工作负载脚本以及初始化测试链的脚本。这一层也定义了配置与结果数据的格式，使上下两层共享数据的格式保持一致。

在开发过程中，上下两层都要用到中间层的配置信息与结果信息，由中间层定义数据格式，可以使上下层更专注自身逻辑的开发，中间层也包含一些区块链启动脚本与初始配置，由下层调用完成启动测试链以及初始化测试等操作。

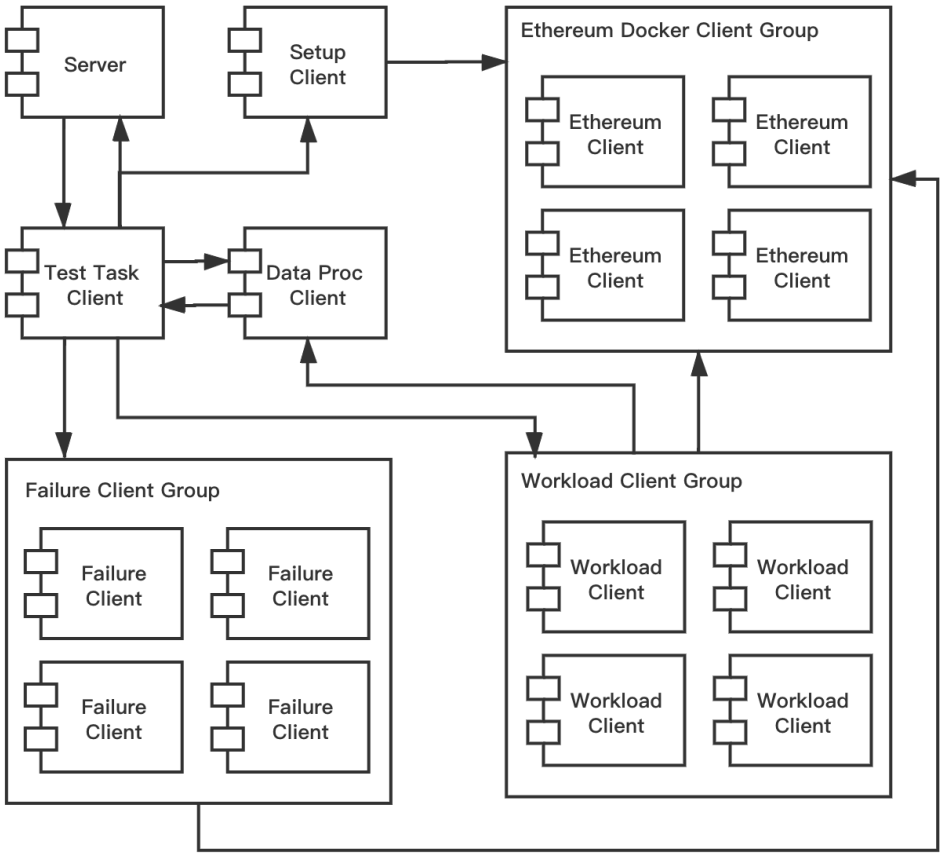


图 3.5: 进程视图

进程视图。图3.5展示了以太坊私有链性能测试系统在执行性能测试任务时的进程视图。进程视图在逻辑视图的基础上展示了一部分服务及实体与进程的对应关系以及进程之间如何通信。如图3.5所示，在服务端主进程收到测试请求后，整个测试任务将完全交给测试任务进程执行，测试配置以 JSON 或 YAML

文件的形式传递给该进程。收到测试开始指令后，测试任务主进程分别启动初始化进程、工作负载进程组、故障注入进程组与数据处理进程。初始化进程根据配置启动以太坊私有链，并初始化其中的挖矿节点、智能合约配置等。初始化进程执行结束后其生命周期终止，工作负载进程组开始向测试链发送请求直到持续时间结束。与此同时，故障注入线程组启动定时器，在预定时间进行故障注入与故障恢复。测试任务完成后，交易数据发送到数据处理进程处理，处理完成的数据逐级返回给服务端进程，交给前端展示。

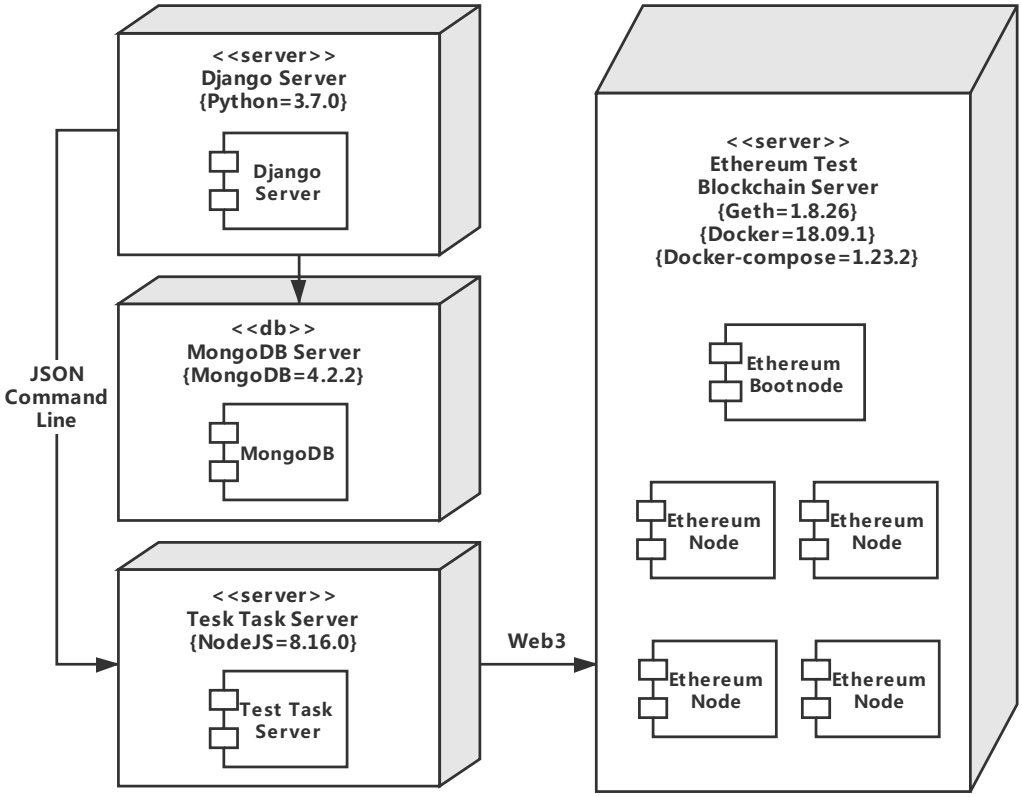


图 3.6: 物理视图

物理视图。图3.6展示了本系统的物理部署视图。物理部署视图主要为运维工程师提供软硬件的对应关系，方便运维管理。图中每一个方块即代表一个物理机，从左到右依次为 Django 服务器、MongoDB 数据库、测试执行模块和以太坊测试链，数据库与 Django 服务器连接用以存取测试任务数据。Django 服务器通过 JSON 文件传输配置，命令行进行调用。测试执行模块通过 Web3 调用以太坊测试链。根据资源分配情况，四部分可以分开部署在不同物理机，也可以部署在单个服务器。由于以太坊私有链消耗资源较多，因此建议分配给部署私有链的物理机较多计算和存储资源。

3.3　系统模块设计

本节根据概要设计中的整体框架设计，进一步划分出的两个模块：测试管理模块与测试执行模块。其中，测试管理模块与前端展示层直接交互，负责处理与存储测试配置与结果信息，并通过 JSON 以及命令行调用的形式与测试执行模块通信；测试执行模块则负责测试链搭建、性能测试任务执行与故障注入等系统核心功能。

3.3.1　测试管理模块

测试管理模块主要负责处理与存储测试配置与结果信息。图3.7为该模块的框架图。本模块主要分为两个子模块：测试配置子模块与结果管理子模块。测试配置子模块负责处理用户输入的配置，解析、验证配置，并根据配置执行测试任务。结果管理子模块则是在测试任务完成后，对测试结果数据进行分析存储。在测试管理模块运行过程中，模块首先从前端展示或数据库中获取用户的测试配置，读取并解析该配置，如发现配置信息不符合约束条件则要求重新输入。然后，解析完成的配置文件将以 JSON 或 YAML 文件格式存储，由测试执行模块进行读取执行。最后，如测试任务执行成功则处理测试结果并存储到数据库中，如执行失败则直接结束本次测试任务。

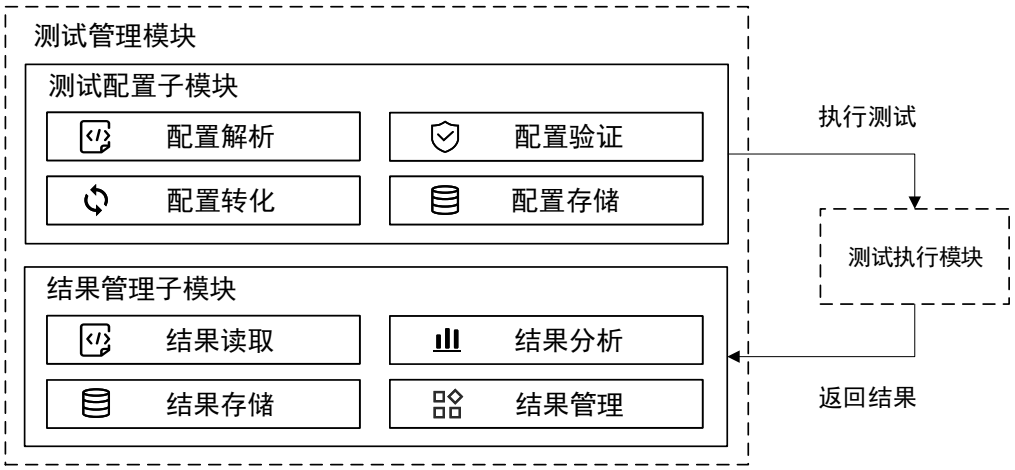


图 3.7: 测试管理模块架构

由于本模块需要存储的测试配置与测试结果的数据结构复杂，数据的层级结果较多，并且模块之间传递信息均以 JSON 格式，因此本模块采用非结构化数据库 MongoDB 进行数据存取。本系统需要存储的信息除用户信息之外，主要包括测试配置信息与测试结果信息。为方便查询与展示测试配置与结果的对应关系，测试配置信息与测试结果信息采用“一对一”的对应关系，将测试配置信息

与测试结果信息合并为一条信息进行存储，测试结果信息作为测试配置信息的一个字段。具体的持久化设计如表3.7所示。

表 3.7: 测试任务字段设计

字段名	类型	描述
id	string	测试任务唯一识别 ID
difficulty	long	以太坊私有链挖矿难度
gasLimit	long	以太坊私有链区块 Gas 限制
nodeCount	int	以太坊节点总数
startUpType	string	以太坊客户端启动方式，如 docker
clientType	string	以太坊客户端类型，如 geth
minerCount	int	挖矿节点总数
startTps	int	请求速率
duration	int	测试持续时间
smartContract	string	工作负载智能合约
failure	array	故障配置，见表3.8
status	string	测试任务执行状态
startTime	timestamp	测试任务开始时间
result	array	测试结果，见表3.9

表3.7展示了测试任务的持久化设计，其中包括区块链启动配置、性能测试与故障配置、测试时间与结果等信息。目前，startUpType、clientType 等字段只有唯一值，如 startUpType 为 docker，clientType 为 geth。为了保证良好的可扩展性，将这两个字段加入测试任务配置，以便于后续支持新的启动方式与客户端类型。另外，failure 字段与 result 字段为嵌套的 JSON 格式文档，下面将介绍这两个字段。

表 3.8: 故障配置字段设计

字段名	类型	描述
type	string	故障类型
startAfter	int	故障开始时间
duration	int	故障持续时间
param	int	故障参数
label	string	故障唯一标签
nodeName	string	故障注入节点
contractName	string	故障负载智能合约

表3.8展示了故障配置部分的持久化设计，对应测试任务字段中的 failure 字段。该部分包括故障的类型，开始语持续时间以及根据不同类型故障的特有字段。其中，共识类型故障中不存在故障严重程度划分，因此无 level 字段。nodeName 字段在网络故障与共识故障中生效，用以确定故障注入的目标节点。contractName 字段只在智能合约故障生效，用以指定故障合约名称。

表 3.9: 测试结果字段设计

字段名	类型	描述
throughput	int array	吞吐量
latency	double array	交易平均延迟

表3.9展示了测试结果部分的持久化设计，对应测试任务字段中的 result 字段。该部分主要包括吞吐量、延迟等测试指标的秒级数据。其他如最大、最小、平均值等指标，可以通过存储的数据计算得出。

3.3.2　测试执行模块

测试执行模块是本系统的核心模块，负责测试链搭建、性能测试任务执行、故障注入等功能。根据测试任务执行的流程，将本模块划分为三个子模块：测试准备子模块、性能测试子模块、故障注入子模块。

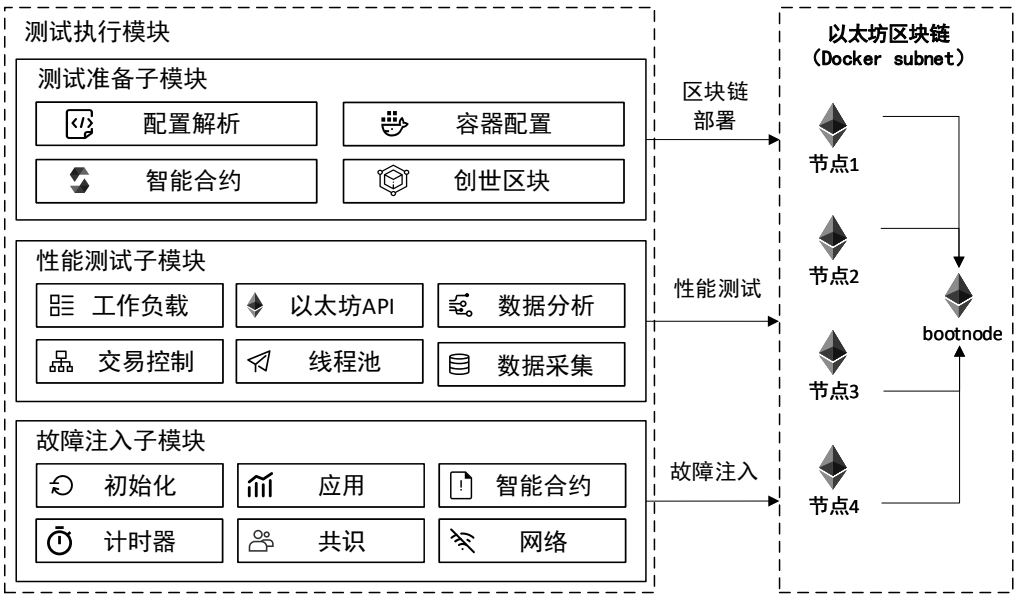


图 3.8: 测试执行模块架构图

如图3.8所示，整个系统分为三个子模块，这三个模块分别于以太坊区块链进行交互，执行区块链部署、性能测试与故障注入任务。测试准备子模块主要用于部署一个待测区块链，其中包括对从测试管理模块获取的测试配置进行解析，并根据配置启动以太坊客户端容器、构建以太坊创世区块、部署智能合约。性能测试子模块则根据配置中的性能测试部分，以异步操作的方式使用以太坊 API 批量发送请求。性能测试结束后收集请求返回结果作为计算指标的数据源。故障注入子模块主要为故障初始化与计时器，以及故障注入的具体实现 [52]。故

障注入子模块一共实现了四种类型的故障：应用故障、共识故障、智能合约故障、网络故障，在表2.2已进行介绍。每种故障分别实现注入方法和恢复方法。此外，每种故障定义了故障参数（共识故障没有参数划分，因为它只有两个状态）来模拟故障的严重性。

在测试过程中，首先，模块使用测试配置来构造测试区块链，包括创世块，docker 配置等。然后，部署智能合约。同时，故障注入层初始化故障。对于每个以太坊客户端，并派生一个线程并将请求发送到客户端。当到了预设时间时，故障注入层将注入故障。最后，收集测试结果并生成性能报告。最终结果包含以秒为单位的测试区块链的性能指标。

3.4　本章小结

本章主要介绍了系统的需求分析与概要设计。首先从需求分析入手，介绍系统的涉众，并给出了系统的用例，在此基础上给出功能、非功能需求。之后，根据需求分析进行了系统概要设计，介绍给出系统的整体架构，并给出系统的4+1 视图。最后，本章给出了系统的模块设计，将系统整体划分为两个模块，测试管理模块与测试执行模块，分别进行概要设计。

第四章　详细设计与实现

根据第三章进行需求分析与概要设计，本章将介绍系统的详细设计与实现。本章对系统的两个模块划分出的五个子模块，即测试管理模块中的测试配置子模块、测试配置子模块，以及测试执行模块中的测试准备子模块、性能测试子模块、故障注入子模块。首先，给出每个子模块的详细设计，介绍其中使用的关键技术，并给出模块的类图与顺序图。其次，介绍核心功能的代码实现。最后，通过系统界面截图与测试过程的系统监控对本系统进行展示。

4.1　测试管理模块

4.1.1　测试管理模块设计

测试管理模块为用户与测试执行模块之间的过渡，主要负责接收用户的请求与展示数据，调用测试执行模块启动测试与获取结果。该模块根据模块的功能主要分为测试配置子模块与结果管理子模块，分别处理用户配置输入与测试结果输出。测试管理模块需要提供界面供用户访问，并且需要提供测试配置与结果存储服务，因此该模块本质上设计为一个提供前端页面的 Web 服务器，以便于用户通过浏览器进行访问。

由于测试结果数据量大，并需要进行复杂的处理操作，因此选择常用于处理数据的 Python 语言，以及支持 Python 语言的 Django[1]服务器。该服务器采用 MVC 架构，其中的 view.py 作为 Controller，负责处理用户请求，其中的函数通过 urls.py 映射到请求 URL。请求处理完成后 Controller 组装好 Model，使用 render 函数对 Django 的 Html 模板进行渲染，生成 View 视图返回给浏览器 [53]。

由于测试配置与结果需要反复查看和使用，二者会以一对一的关系保存在数据库中。测试配置本身以 JSON 或 YAML 的格式传递，字段之间层级嵌套较多，而且数据没有固定的结构，属于非结构化数据。测试结果数据量大，每一个指标数据结果通常会超过 1000 条，对数据库的读取性能要求较高。因此，关系型数据库存储不适用于本系统。MongoDB[2]是一种非关系型数据库，具有比关系型数据库更为快速的读写性能，支持 JSON 格式数据 [54]。由于测试配置与测试结果数据均为 JSON 结果，因此选择 MongoDB 作为数据库。

[1]Django. https://www.djangoproject.com/

[2]MongoDB. https://www.mongodb.com/

（1）测试配置子模块

测试配置子模块负责读取、验证并生成测试配置。该模块从用户处获取测试配置信息，解析并验证配置的合理性，最后转化为可以被测试执行模块识别的配置格式。测试配置信息包括三种配置，即区块链配置、性能测试配置与故障注入配置。区块链配置为以太坊私有链启动配置与创世区块的配置，其中包含 Difficulty 与 Gas Limit 设置。性能测试配置包括测试的请求速率、持续时间以及工作负载智能合约。故障配置包括故障类型、参数、开始、持续时间以及每种故障各自的配置。用户通过页面输入以上配置，服务端读取配置并生成配置文件，通过命令行以文件为参数启动测试。

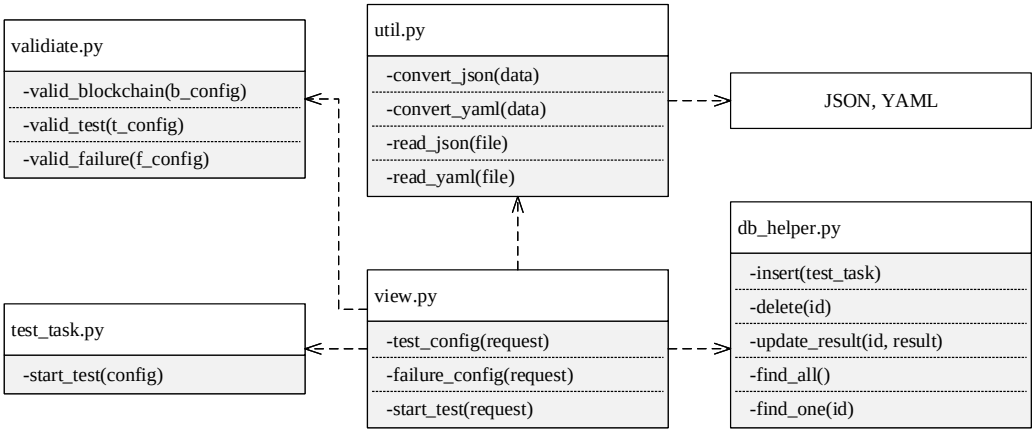


图 4.1: 测试配置子模块类图

如图4.1所示为测试配置子模块的类图。测试配置的输入通过前端页面进行，view.py 为 MVC 框架中的 Controller，用户输入的配置将通过请求 URL 映射到 Controller 中的视图函数进行处理。validate.py 为配置验证服务，用于验证三种配置文件的合法性，只有三次验证均通过才允许保存与执行。由于需要生成配置文件，util.py 提供了 JSON 与 YAML 格式的文件转换服务，该功能在其他模块中使用频率较高，因此将其作为工具类单独抽离出一个服务。由于配置信息需要展示与反复使用，测试配置将以 JSON 数据格式保存在 MongoDB 数据库中，数据的增删改查通过封装的 db_helper.py 执行。start_test.py 负责调用测试执行模块，其中封装了用于启动测试任务的命令行。

如图4.2所示为测试配置子模块的顺序图。用户按顺序依次输入区块链配置、性能测试配置与故障注入配置后，系统依次对配置进行验证。如果所有配置验证通过，则系统会首先将测试配置写入文件，然后通过命令行与配置文件地址启动测试执行模块开始测试任务。由于一般性能测试持续时间长，系统采用异步的方式启动测试，系统先返回测试任务开始执行的信息，此时该测试任务为

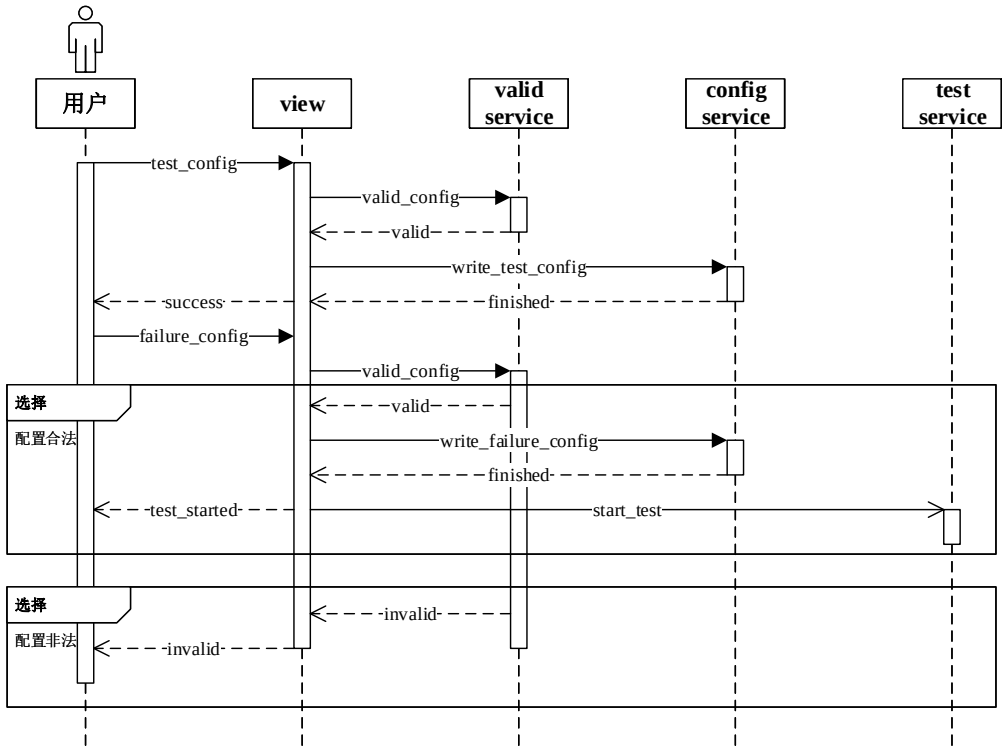


图 4.2: 测试配置子模块顺序图

执行中状态。在测试结束后，该测试任务的状态自动变更为已完成，此时可以查看测试结果。如果测试配置验证非法，则无法执行测试，配置信息也不会存储到数据库中，系统会通知用户配置错误，要求重新输入。

（2）结果管理子模块

结果管理子模块主要负责处理测试任务执行完成后产生的测试结果数据，并对处理结果进行展示。在测试执行模块完成测试任务后，会生成包含秒级性能指标的测试结果 JSON 文件。结果管理子模块会读取这些文件中的数据，与测试配置一同存储在 MongoDB 数据库，以便后续查看。结果展示时，本模块使用 Python 进行数据格式转换，并使用 NumPy[3]进行数据处理。NumPy 是 Python 中功能完整且应用广泛的数据处理包，可以实现多维数组切片、矩阵运算等功能[55]。测试结果使用 NumPy 对空值、零值进行数据清洗，计算吞吐量、延迟等指标的平均值、中位数等。

由于区块链打包区块速率的不确定性，每产生一个区块就会有大量的交易同时返回。由于区块产生的时间间隔较长，整个测试过程中的秒级性能指标随着区块的产生有较大的抖动。针对这种问题，本模块中实现了均值滤波算法，根

[3]NumPy. https://numpy.org/

据不同的指标类型与数据量大小，对结果数据进行不同程度的平滑处理。在每次展示指标图表时，对原始数据使用滤波算法减少数据波动，以便于展示数据的整体趋势。然而滤波算法可能对部分指标的峰值产生影响，系统也保留了原始数据展示的功能。

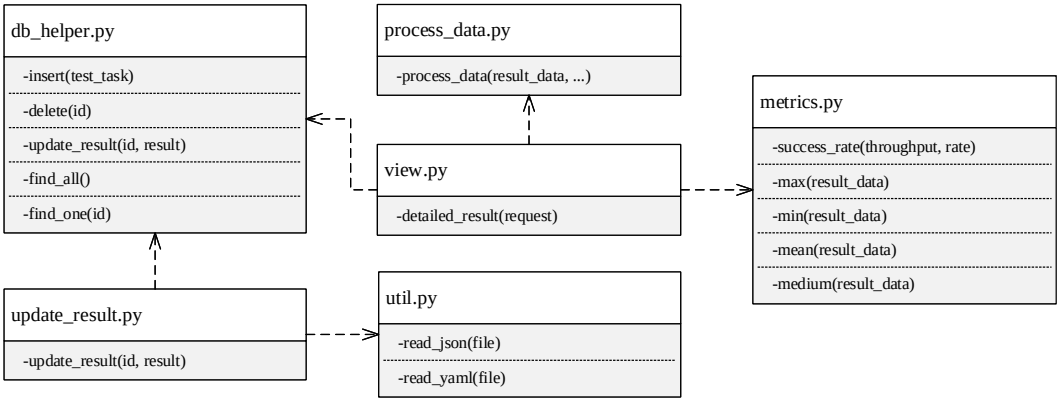


图 4.3: 结果管理子模块类图

如图4.3所示为结果管理子模块的类图。其中，load_result.py 实现了定时任务，每个一段时间通过 util.py 的存取文件服务拉取测试结果，并直接存储到数据库中，数据的存取复用 db_helper.py。metrics.py 为指标计算服务，使用 Python 的 NumPy 包计算性能指标。该服务用于计算整个测试过程中、故障过程中以及故障注入前与恢复后的性能指标平均值、中位数等。data_process.py 为数据处理服务，对不同类型的指标进行不同程度的平滑处理，滤波算法结合 SciPy[4]包实现。指标结果折线图通过 ECharts.js[5]在 H5 页面展示。用户也可以选择展示未经处理的测试结果，通过在 URL 请求参数中添加结果类型字段 type 实现，参数为 raw、smooth，分别代指原始、平滑数据。

$$smooth\_metric_t = \frac{\sum_{i=t-n}^{t+n} raw\_metric_i}{2n+1} \tag{4.1}$$

滤波算法采用平均值滤波法，如公式4.1所示。$smooth\_metric_t$ 为进行平滑处理后的第 t 秒的指标数据。该算法对第 t 秒的指标数据取包括自己在内前后各 n 个数据，共 $2n+1$ 个数据的平均值作为第 t 秒的平滑指标数据。这种滤波法可以保证数据更加平滑，对一些由于区块产生间隔造成的指标突增与突降进行过滤，使数据的增长与下降趋势更为明显。同时，该算法也保证了所有秒级数据总和

[4]SciPy. https://www.scipy.org

[5]ECharts.js. https://echarts.apache.org/en/index.html

不变，对吞吐量指标而言，不会改变整体测试过程中返回的请求数量，也就不会影响人们对本次测试整体吞吐量的认知。

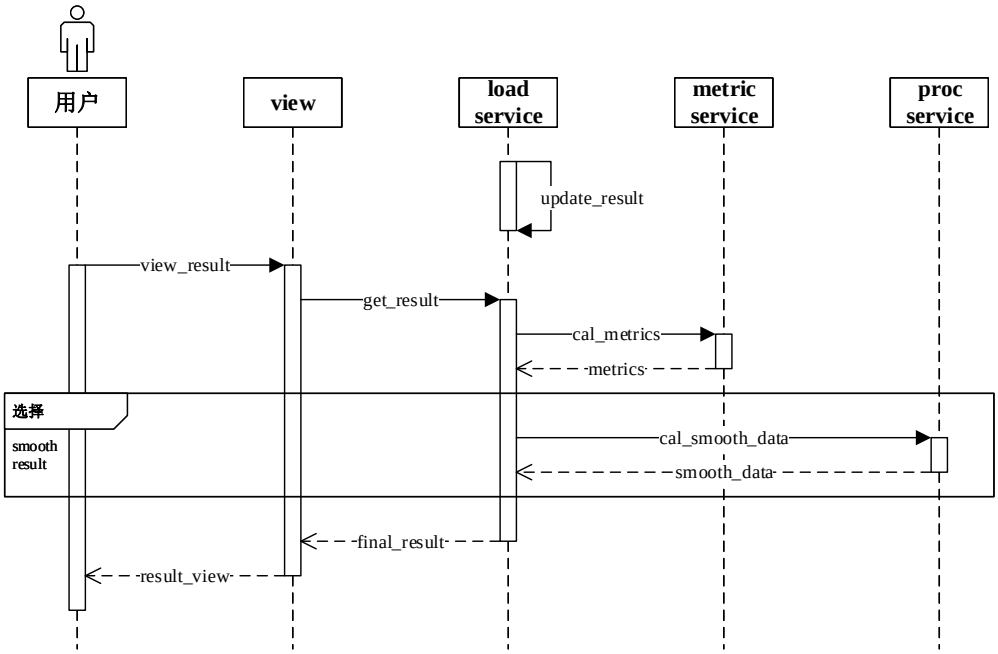


图 4.4: 结果管理子模块顺序图

如图4.4所示为测试任务执行完成后，展示处理完成的测试结果流程的顺序图。每个测试任务的测试结果会在执行完成后，有存储测试结果的定时服务读取测试执行模块产生的结果文件，并存储在 MongoDB 数据库中，随后结果文件将被清理。测试结果与其对应的测试配置将置于一个 JSON 对象中存入数据库，以便根据配置进行查找。在用户查看结果时，系统根据测试任务 ID 从数据库中读取测试结果，首先使用原始数据进行指标计算，之后对数据进行滤波处理。处理后的数据将交给视图控制器进行封装，最终展示给用户。

4.1.2　测试管理模块实现

（1）测试配置子模块

图4.5给出了 view.py 中输入性能测试配置与故障注入配置视图的代码实现。由于这故障注入配置依赖区块链配置提供的节点信息，执行该部分代码需要用户首先输入区块链配置，以确定哪些节点可以注入故障。用户提交的请求后，请求信息会以参数的形式传入该函数。第 4 行中，首先判断是否为 POST 请求，并从请求中获取性能测试配置参数，如请求速率、持续时间等。之后系统读取之前生成的区块链配置，并与性能测试配置整合。第 12、15 行分半验证性能测试与故障注入配置的合法性，如果验证非法则返回配置非法提示页面。待所有配

```
1.  @login_required
2.  def test_config(request):
3.      username = request.user.username
4.      if request.method == "POST":
5.          start_tps = int(request.POST.get("startTps", None))
6.          duration = int(request.POST.get("duration", None))
7.          ...
8.      test_config = util.read_json('static/json/config.json')
9.      test_config['user'] = username
10.     ...
11.     test_config['startTime'] = int(time.time())
12.     if not validate.validate_test_config(test_config):
13.         return render(request, 'testConfig_py.html'
                          , {'err_msg': 'Invalid test config.'})
14.     failure_config = [...]
15.     if not validate.validate_failure_config(failure_config):
16.         return render(request, 'testConfig_py.html'
                          , {'err_msg': 'Invalid failure config.'})
17.     util.write_yaml('static/json/failure.yaml', {'failure': failure_config})
18.     test_config['failure'] = [failure_config]
19.     util.write_json('static/json/config.json', test_config)
20.     db_helper.insert(test_config)
21.     return render(request, 'testConfig_py.html', {'msg': 'All config valid.'})
```

图 4.5: 测试配置子模块代码实现

置验证通过后，将三种配置整合为一个 JSON 对象，作为本次测试任务的配置写入 JSON 配置文件，并于第 20 行通过 db_helper.py 将配置保存到数据库中。同时，为方便本模块调用测试执行模块，第 17 行将故障注入配置单独写入 YAML 文件，作为参数写入命令行。

（2）结果管理子模块

图4.6为 data_process.py 中，指标数据的数据清洗以及滤波算法实现。函数传入的参数 data 为待处理数据，结构为二维数组。数组第一列为时间，代表测试开始后的时间点，单位为秒，第二列为该秒的性能指标。第 4 行通过 NumPy 将数据纵向切为时间和指标两部分。由于测试过程中可能存在某些时段无正常请求返回，此时平均交易延迟指标为空值。为了不影响整体的数值计算，第 5-8 行进行数据清洗，将空值填充为前一个邻近值。第 9 行利用 SciPy 提供的平均值滤波算法对数据结果进行滤波。而对于滤波后的数据，由于其仍有小范围内的数据抖动，为了更好的展示数据趋势，系统对数据进行间隔取样，同时对时间进行同样的间隔取样。在第 11-14 行，对于交易延迟数据中由于无交易返回造成的

```python
def proc_data(data, mean, interval):
    res = []
    tmp = np.array(data)
    x, y = tmp[:, 0], tmp[:, 1]
    for i in range(len(y)):
        if y[i] is None:
            y[i] = 0 if i == 0 else y[i - 1]
        res.append(int(y[i]))
    y = signal.medfilt(res, mean)
    x, y = x[0::interval], y[0::interval]
    for i in range(1, len(x) - 1):
        if y[i] == 0:
            if y[i - 1] != 0 and y[i + 1] != 0:
                y[i] = (y[i - 1] + y[i + 1]) / 2
    d = np.dstack((x, y))
    return d[0].tolist()
```

图 4.6: 结果管理子模块-数据处理代码实现

0 值，因其可能影响整体平均值，将该 0 值取前后值的平均值。最后将两个数组重新合并，组成最终处理完成的结果。

4.2　测试执行模块

4.2.1　测试执行模块设计

测试执行模块为本系统的核心模块，负责系统的执行测试任务。该模块接收测试配置子模块生成的测试配置、解析并完成测试链搭建与初始化、性能测试执行、故障注入。测试任务结束后，收集所有请求的返回结果，计算返回的交易数量与每个交易的返回时间。根据功能系分，将该模块分为三个子模块：（1）测试准备子模块，负责以太坊测试链搭建，智能合约部署以及故障初始化；（2）性能测试子模块，负责以配置的速率向测试链发送请求，收集请求返回数据；（3）故障注入子模块，负责根据配置注入与恢复故障。整个测试流程需要较多耗时长的异步操作，如测试链启动、智能合约部署、请求发送与返回值处理、故障注入等，因此选择支持异步操作较好的 NodeJS[6]作为该模块的实现。利用 NodeJS 的 Promise 机制，可以通过 async、await 等关键字，实现同步异步操作的切换。该模块设计为命令行形式调用，可由测试管理模块进行调用，也可单独执行。调用时区块链、性能测试、故障注入配置文件地址为参数，完成调用后测试结果文件生成在预设置的目录下。

[6]NodeJS. https://nodejs.org/

（1）测试准备子模块

测试准备子模块主要处理开始执行测试任务之前所需的准备工作。在模块读取区块链配置后，首先通过 Docker[7]启动以太坊测试链。Docker 是一种容器技术，可以通过预先打包好的镜像短时间启动容器，每个容器类似于一个小操作系统单独运行一个进程，可以良好的隔离不同容器的运行环境 [56]。Docker-compose[8]是一个用于快速搭建多个 Docker 容器，并使容器间迅速建立起关联的工具。该工具可以通过一个简单的配置文件按顺序启动多个以太坊客户端容器，然后搭建起一个小型的局域网，为每个节点分配 IP 地址，并在其中运行以太坊测试链 [57]。Docker 中还可以进行文件夹挂载，这种方式可以将用户配置好的以太坊初始区块配置挂载到每个以太坊节点之中。这样可以不用涉及到容器内部操作，修改被挂载的文件夹中的内容即可根据用户配置启动以太坊客户端。每个节点的启动脚本与初始账户文件也同样挂载到容器中。测试结束后，容器可以自动销毁，不会余留生成区块的数据，而挂载的文件仍然保留。测试链搭建完成后，系统还会进行智能合约部署以及故障初始化等步骤。在本模块所有步骤完成后，才可以开始性能测试。

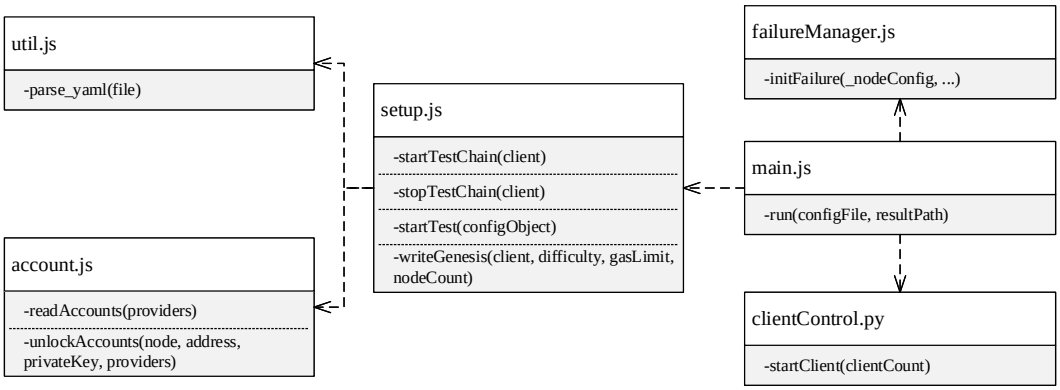


图 4.7: 测试准备子模块类图

如图4.7所示为测试准备子模块的类图。该模块的关键功能实现为 setup.js，被 main 函数直接调用。setup.js 中包含启动与终止以太坊测试链的函数，由 NodeJS 中的 docker-api 与 docker-compose 包实现。系统预设置了一个 docker-compose.yaml 文件，用以快速启动以太坊测试链。对于启动的每一个以太坊节点，均有一个文件夹在启动时挂载到该节点中，用以控制初始区块的配置与启动脚本。初始区块配置根据用户输入的区块链配置直接写入该文件夹。每个节点都预置了账号文件，所有账号在以太坊启动后处于锁定状态，需要调用 account.js

[7]Docker. https://www.docker.com/

[8]Docker-compose. https://docs.docker.com/compose/

手动进行解锁才可以部署智能合约和发送交易请求。setup.js 会在账户解锁后部署智能合约，以全局变量的形式存储，但只可读不可更改。故障也需要在测试开始前进行初始化，通过 failureManager.js 执行。性能测试所需的多线程的线程池在 clientControl.js 中启动。

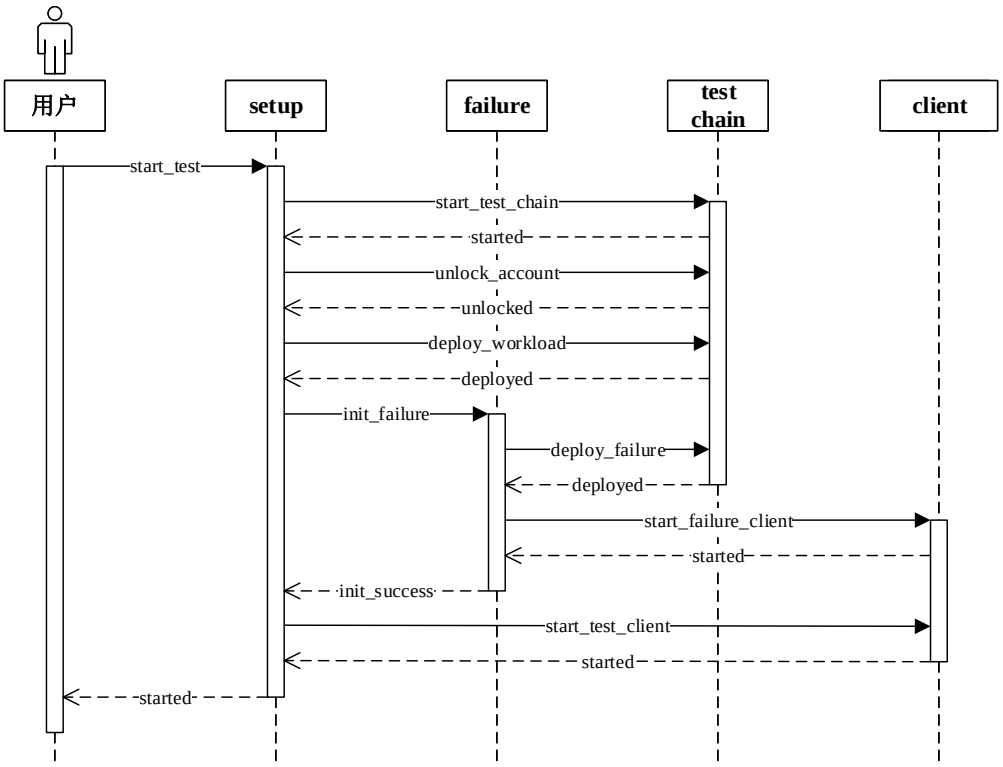


图 4.8: 测试准备子模块顺序图

如图4.8所示为测试准备阶段的顺序图。当用户开始测试任务，启动服务将直接启动以太坊测试链。Docker 会首先启动一个 bootnode，用于搭建所有以太坊节点的共识网络。之后，以 bootnode 为连接共识网络的初始节点启动剩余以太坊节点。每个节点会内置启动脚本，脚本中包含该节点是否启动挖矿的代码。非挖矿节点启动后即可开始验证区块，挖矿节点启动后需要进行预热。预热过程根据服务器性能持续数十秒到数分钟，过程中主进程会暂时挂起等待其预热完成。在所有以太坊节点成功启动后，主进程通过以太坊的 web3.js[9]解锁账户，并使用解锁后的账户部署作为工作负载的智能合约，将部署后的地址记录到 contractInfo 中。之后，系统开始初始化故障。负责初始化故障的服务将部署智能合约故障中的低性能合约，启动故障注入线程池。最后，系统初始化性能测试线程池，确定每个线程对应的以太坊节点，完成整个测试准备阶段。

[9]web3.js. https://github.com/ethereum/web3.js/

（2）性能测试子模块

性能测试子模块在准备任务结束后开始运行，主要负责按照设定的负载发送交易请求，收集并处理返回数据。模块以线程控制器为控制中心，通过向预先启动好的线程发送开始测试的消息，所有线程将开始向每个以太坊节点发送交易请求，每个节点对应一个线程。线程使用以太坊团队官方提供的 web3.js 作为与以太坊客户端交互的 API。针对所有可能的 API 返回结果，系统将其划分为成功与失败两种类型，只有成功的返回才会被计算吞吐量与延迟指标。每个请求会设置一个超时时长，超时的请求也被认为是失败请求。最终，每个线程会记录每个节点的返回结果的数据，统一传给主线程进行数据处理。主线程会获取每个请求结果的发送和返回时间、是否为成功请求、是否超时等信息，计算出每秒的性能指标，得出最终的测试结果。

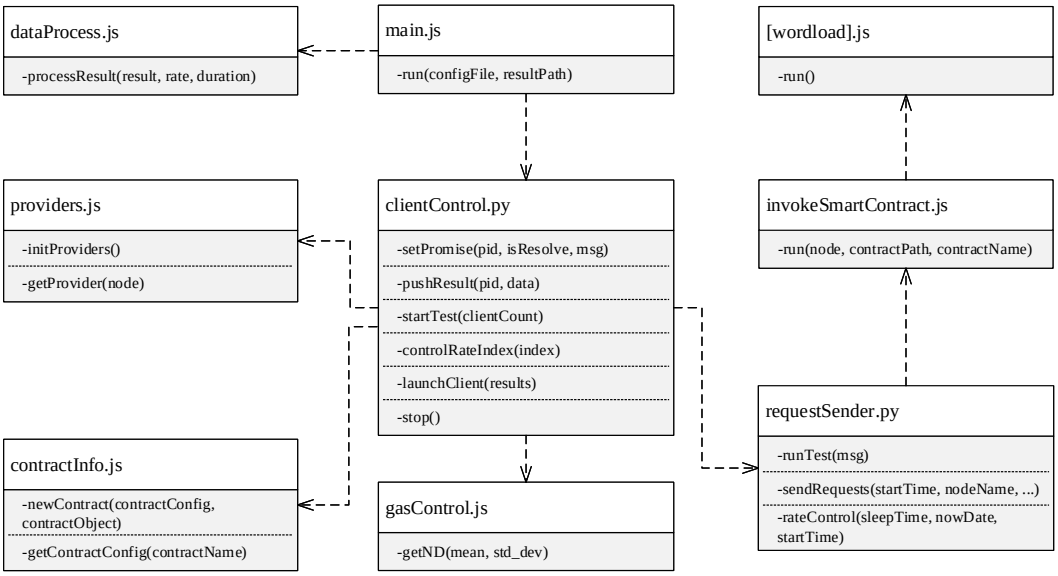


图 4.9: 性能测试子模块类图

如图4.9所示为性能测试子模块的类图。性能测试子模块的核心为线程控制器 clientControl.js，通过线程通信的方式启动与控制线程。线程实例为 request-Sender.js，其中实现了发送请求与回传数据的功能。线程实例可以接收线程控制器的开始、停止测试或变更请求速率等消息类型。每个线程测试结果直接返回给控制器进行整合计算。测试结束后控制器将结果数据返回给主线程，主线程交给 dataProcess.js 处理，计算秒级的性能指标。发送请求所需的工作负载智能合约存储在 contractInfo.js 中，其中的合约信息在部署后生成。与以太坊客户端交互的 API 被封装在 invokeSmartContract.js 中，在每个线程发送请求时调用。由于不同智能合约有不同的参数，每个工作负载合约会有需要对应的 JS 脚本用于生成请

求参数。系统提供了 HelloWorld 智能合约，参数生成函数对应 HelloWorld.js 的 run 函数。针对不同使用场景，用户可能会部署自己实现的合约，请求参数的生成方式也可由用户自行设计。

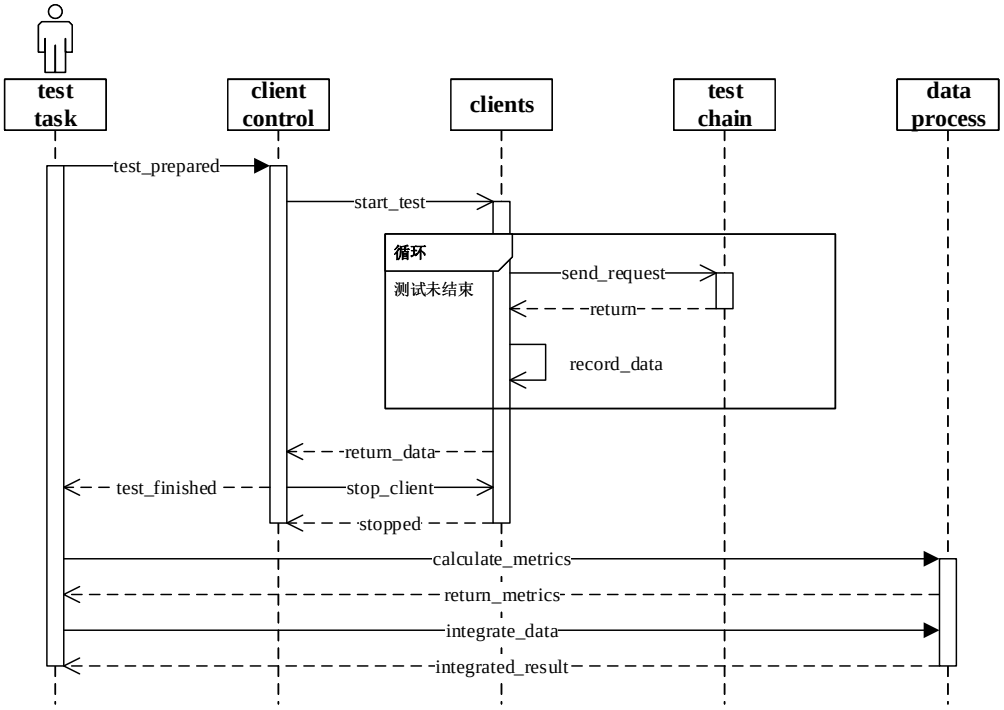


图 4.10: 性能测试子模块顺序图

如图4.10所示为性能测试过程的顺序图。测试准备工作完成后，系统首先会根据配置启动线程控制器。控制器向每个线程发送开始测试的指令，包括对于节点的代理、请求速率、持续时间等。收到指令的线程开始循环发送请求，根据请求速率计算出每两个请求之间的时间间隔，发送间隔使用 sleep 函数暂时挂起线程。所有请求的发送均为异步，发送完成后该异步请求继续等待返回，如果超时直接记为失败请求，而主线程继续发送请求直到测试计时结束时。最后，每个线程等到所有请求返回或超时，将请求结果回传给主线程。主线程再通过数据处理服务计算性能指标并聚合数据，生成最终的测试结果。

（3）故障注入子模块

故障注入子模块负责在性能测试执行过程中注入故障，主要由故障控制器与四类故障注入的实现组成。故障注入子模块首先会通过测试准备模块对待注入的故障进行初始化。随着性能测试的开始，故障计时器开始计时，当到达设定的时间时触发故障注入事件。本模块一共实现了四种类型的故障，分别是应用、共识、智能合约与网络。四种故障由于属于不同类型，因此实现方式完全不同，

其中除共识故障外，每种故障有可以配置参数以表示故障的严重程度。应用故障为流量波动故障，该故障由控制每个线程的请求速率实现。共识故障为节点的退出与加入，通过对 Docker 容器进行暂停和重启实现。智能合约故障则实现了一个低性能的智能合约，通过将一半的正常请求转化为低性能智能合约请求实现。网络故障则通过一个 Docker 的混沌工程工具 Pumba[10]实现，该工具可以对处于局域网中 Docker 容器注入网络丢包和延迟。

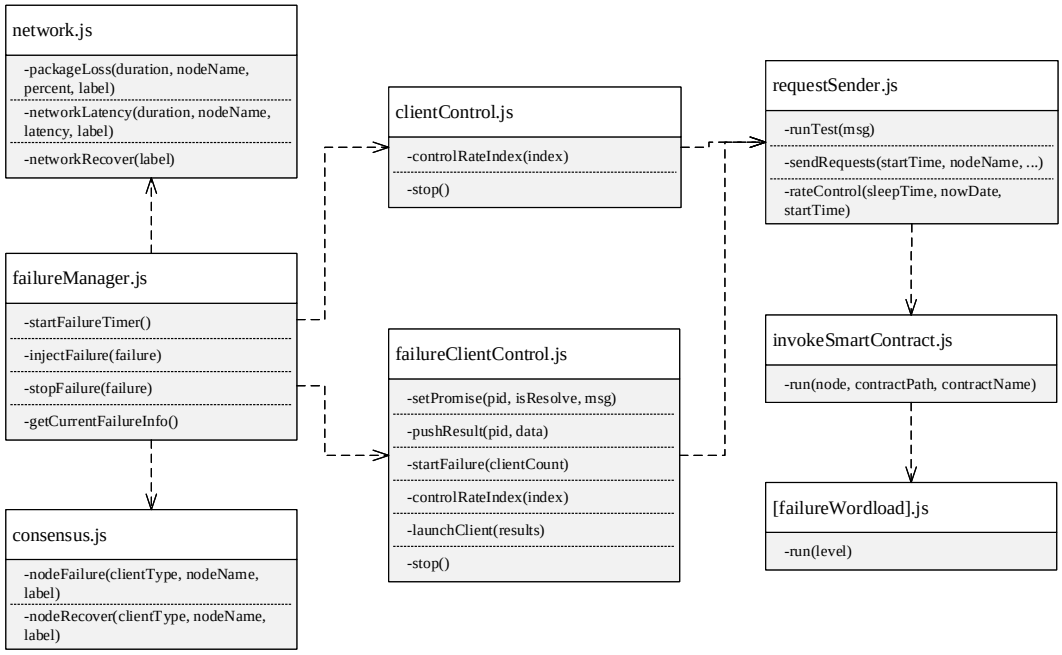


图 4.11: 故障注入子模块类图

如图4.11所示为故障注入子模块的类图。故障注入由故障管理器 failureManager.js 进行统一管理。故障管理器将所有类型故障注入操作进行参数化，统一抽离出故障注入与故障恢复方法，方便统一调用与故障类型的扩展。所有故障类型都有注入时间、持续时间与故障参数等。故障控制器也实现了计时功能，性能测试开始时为每一个故障启动注入与恢复倒计时，在到达预设的时间时自动调用故障注入与恢复函数。故障的实现分为两种，一种是直接调用 API 可以实现的，如共识和网络故障。对于二者分别封装了 consensus.js 与 network.js 两个故障注入实现，分别实现了注入与恢复方法，调用参数则由用户在故障配置中提供。对于涉及到请求速率与交易对象的故障，故障的注入与恢复则涉及到与线程的通信。应用故障可以通过线程控制器控制请求速率实现，通过调用线程控制器向线程发送一个请求速率变化倍率的信息，通过变化倍率重新计算每两个请求之

[10]Pumba. https://github.com/alexei-led/pumba

间的间隔时间实现流量激增。而智能合约故障则是实现了一个低性能的智能合约，原请求速率降为之前的 50%，另 50% 的请求转化为 failureClientControl.js 的低性能合约调用请求，实现智能合约故障注入。故障恢复时则将请求速率恢复，并停止 failureClientControl.js 控制的线程。

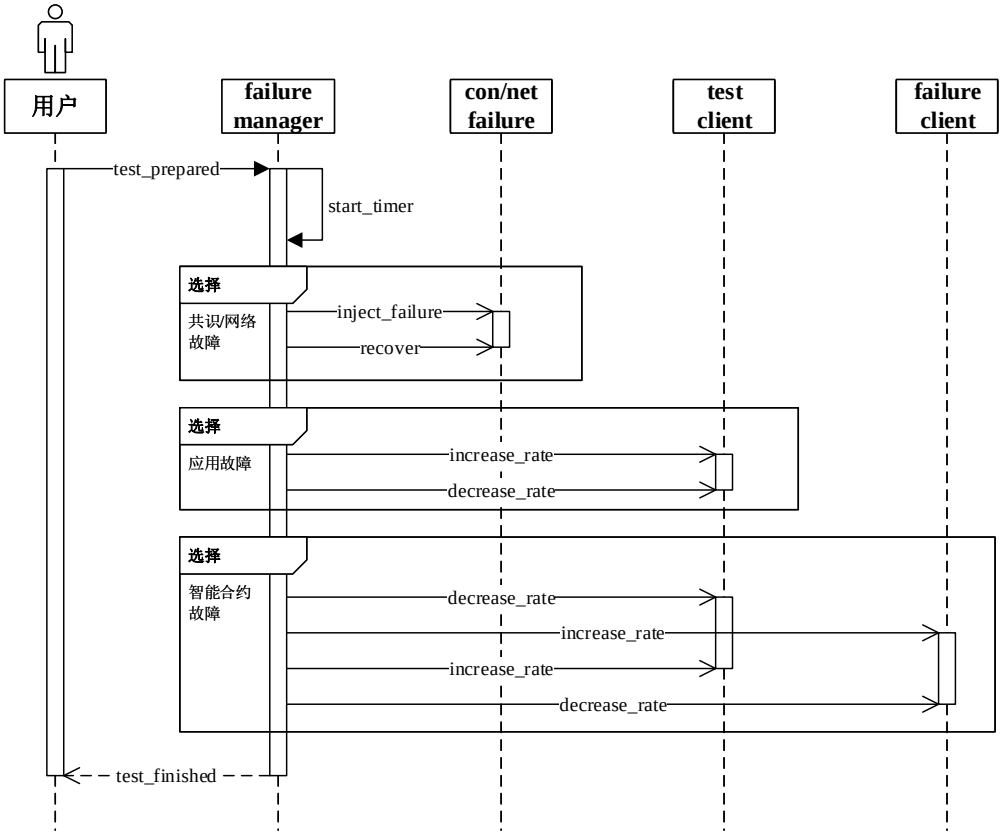


图 4.12: 故障注入子模块顺序图

如图4.12所示为故障注入时的顺序图。首先，在性能测试启动后，该模块立刻启动故障注入计时器。计时器会启动为所有需要注入的故障启动倒计时，即在预设的故障注入时间后调用故障注入函数，在故障持续时间结束后调用故障恢复函数。不同的故障类型具有不同的注入流程。共识与网络故障只需要调用 API 即可完成注入与恢复故障，因此这两类故障注入的被单独封装成服务，只需要在故障注入与恢复时调用对应服务。应用故障与智能合约故障与系统发送的请求有关，因此需要通过操作发送请求的线程实现。应用故障对应流量增长，因此只需调高一次请求速率。智能合约故障需要改变请求对象，注入时会通过与应用故障相同的方式调高故障请求的速率，同时也要降低普通的请求的速率，保证与正常请求速率一致。故障恢复时将上述操作复原即可。

4.2.2　测试执行模块实现

```
1.  module.exports.start = async function (configObject) {
2.      let nodesConfig = configObject.nodes;
3.      await providers.initProviders(nodesConfig);
4.      let promises = [];
5.      for (let i = 0; i < configObject.contracts.length; i++) {
6.          promises.push(install.run(nodesConfig[0].nodeName, configObject.contract
    s[i].path, configObject.contracts[i].name));
7.      }
8.      let contracts = await Promise.all(promises);
9.      for (let i = 0; i < configObject.contracts.length; i++) {
10.         contractInfo.newContract(configObject.contracts[i], contracts[i]);
11.     }
12. };
13.
14. module.exports.startTestChain = async function startTestChain(client) {
15.     winston.info(`Start test blockchain, client type: ${client} ...`);
16.     if (client === 'geth') {
17.         let result = await dockerCompose.upAll({cwd: path.join(__dirname, client
    , 'docker'), log: true});
18.         return result.exitCode;
19.     }
20.     return 1;
21. };
```

图 4.13: 测试准备子模块-准备阶段代码实现

（1）测试准备子模块

图4.13为测试准备子模块中，setup.js 启动区块链与预部署智能合约的代码实现。第 14-21 行的 startTestChain 函数在测试准备阶段调用，首先以配置好的 docker-compose.yaml 启动以太坊测试链。第 17 行中，由于需要等待测试链完全搭建完成后才能部署合约，因此在调用时，需要通过 await 关键字挂起主线程等待函数。在以太坊测试链搭建完成后，调用 start 函数。start 函数负责初始化 web3.js 的组件，并部署工作负载智能合约。第 2 行的 nodeConfig 为以太坊节点的配置，包含了 IP 地址、端口、通信协议等信息。根据节点信息，系统使用 Web3.js 为每个节点封装一个 Provider 对象，每个 Provider 在开始测试时分配给对应节点的线程用以与节点通信。第 8-11 行为部署智能合约操作。系统可以支持部署多个智能合约，系统通过任意以太坊节点发送部署合约请求，请求返回值包括该合约的地址。等所有合约部署完成后，合约地址、交易函数等信息会记录在 contractInfo 中。

```
1.  version: '3'
2.  services:
3.    node0:
4.      container_name: node0
5.      image: ethereum/client-go:alltools-v1.8.26
6.      command: sh ./work/start_geth.sh
7.      working_dir: /node0
8.      volumes:
9.      - ./tmp/node0/keystore:/node0/keystore
10.     - ./tmp/node0/work:/node0/work
11.     ports:
12.     - 8545:8545
13.     - 30303:30303
14.     networks:
15.       etherNet:
16.         ipv4_address: 172.19.0.2
17.     depends_on:
18.     - bootnode
19. ...
20. networks:
21.   etherNet:
22.     ipam:
23.       config:
24.       - subnet: 172.19.0.0/16
```

图 4.14: 测试准备子模块-docker 配置实现

图4.14为测试准备子模块中，预置的 docker-compose.yaml 文件的部分内容。该文件在测试准备阶段中使用，由 NodeJS 的 Docker API 调用。其中，services 字段中为系统启动的容器服务，以第 3-18 行的以太坊节点 node0 配置为例。容器名称命名为 node0，便于在故障注入时选定注入容器。image 为以太坊客户端镜像版本，这里选择以太坊官方的 Geth 客户端 1.8.26 版本。command 为启动镜像后在镜像内部调用的命令，此处预定义了两个启动脚本，用以区分节点是否为挖矿节点。work_dir 字段为容器内部的以太坊客户端的工作目录。volumes 字段中配置的两个外部文件夹可以挂在到工作目录中，使容器内部也可以使用文件夹中的文件。本配置中 keystore 与 work 文件夹分别对应账户信息与启动脚本。ports 为容器的端口映射，由于以太坊需要同时启动多个节点容器，端口映射可以避免容器之间的端口占用，外部请求可以通过映射后的端口与内部容器进行通信。depends_on 字段表示该需要在 bootnode 容器启动完成后才可以启动。第 20-14 行的 networks 字段为整个 Docker 服务搭建了 172.19.0.0/16 的 Docker 子网。第 14-16 行的 network 字段指定了 node0 容器在 Docker 子网的 ipv4 地址。

（2）性能测试子模块

```
1.  async function startTest (nodeConfig, nodeAccounts, rate, duration, contractConf
    ig, results) {
2.      let promises = [];
3.      let i = 0;
4.      for(let id in processes) {
5.          let msg = {
6.              type: 'test',
7.              nodeName: nodeConfig[i].nodeName,
8.              accounts: nodeAccounts[nodeConfig[i].nodeName],
9.              proxy: nodeConfig[i].proxy,
10.             rate: rate,
11.             duration: duration,
12.             contractConfig: contractConfig
13.         };
14.         let client = processes[id];
15.         ...
16.         client.results = results;
17.         client.obj.send(msg);
18.         i++;
19.     }
20.     ...
21. };
```

图 4.15: 性能测试子模块-线程控制器代码实现

图4.15为性能测试子模块中，clientControl.js 线程控制器通知线程启动测试的代码实现。请求线程在准备阶段已经启动，线程信息存储在 process 结构中作为线程池，线程 id 作为取线程的 key。通过遍历 process 线程池，第 17 行中调用 child_process 组件的 send 函数发送信息到子线程中。第 5-13 行中，线程控制器与子线程约定了一组信息，包括开始测试必要的配置信息。其中，type 字段为消息类型，其中 text 为启动测试信息，子线程接收到 type 为 test 信息则开始执行测试任务；nodeName、account、proxy 为节点信息，分别对应以太坊节点名称、该节点中的预置账户、与该节点通信的 web3 代理；rate、duration、contractConfig 为测试配置信息，分别对应请求速率、持续时间、工作负载智能合约配置。client.result 为测试结果对象的引用，线程在执行测试过程中将测试结果直接置入该对象中。线程控制器也持有该对象的引用，测试结束后主线程可以直接通过该引用获取结果进行处理。

图4.16为性能测试子模块中，requestSender.js 收到启动测试信息后发送请求的代码实现。线程在收到启动测试的消息后，以消息中的参数调用该函数。线

```
1.  async function sendRequests(startTime, nodeName, web3, rate, duration, address,
    abi, contractWorkloadPath, type, param) {
2.      winston.info(`${nodeName}: start time ${startTime}`);
3.      const workloadGeneration = require(path.join(rootDir, contractWorkloadPath))
    ;
4.      let promises = [];
5.      let sleepTime = 1000 / rate;
6.      while ((Date.now() - startTime)/1000 < duration){
7.          let account = getRandomAccount();
8.          let workload = param === null ? workloadGeneration.run() : workloadGener
    ation.run(param);
9.          let func;
10.         for (let i = 0; i < abi.length; i++) {
11.             if (abi[i].name === workload.func) {
12.                 func = abi[i];
13.                 break;
14.             }
15.         }
16.         promises.push(invoke.submitTransaction(nodeName, web3, address, func, ac
    count[0], account[1], workload.param));
17.         txNum += 1;
18.         await rateControl(sleepTime / rateControlIndex, Date.now(), startTime);
19.     }
20.     return await Promise.all(promises);
21. }
```

图 4.16: 性能测试子模块-线程发送请求代码实现

程控制器传递的信息包括请求速率、持续时间、节点配置、请求账户等信息，这些信息作为参数传递进入 sendRequests 函数。contractConfig 为本次测试的工作负载合约配置，第 3 行中，线程根据配置获取智能合约的参数生成 JS 脚本地址，通过 require 关键字的形式读取到 workloadGeneration 对象中。第 6-19 行为性能测试循环发送请求的过程，startTime 为测试开始时间戳，循环中通过 Date.now 函数获取当前时间戳，计算与开始时间戳的差判断是否到达测试持续时间跳出循环。线程首先会根据速率计算两个请求的时间间隔 sleepTime。每次进入循环语句时，第 7-15 行选取随机账户并调用 workloadGeneration 生成交易参数，第 16 行调用 submitTransaction 发送请求。submitTransaction 为异步函数，返回一个 Promise 对象。该 Promise 对象将被放入 promises 的数组中。第 18 行调用 sleep 函数，在 sleepTime 的时间内挂起线程等待下次发送请求。最后使用 Promise.all 函数等待 promises 数组中的异步操作，该函数只有在所有异步执行的 Promise 对象都返回后才会调用完成。

（3）故障注入子模块

```
1.  async function injectFailure(failure) {
2.      winston.info(`Start failure injection: ${failure.label}`);
3.      failure.status = 'injected';
4.      failure.startTime = Date.now();
5.      switch (failure.type) {
6.          case 'application-normal':
7.              clientControl.controlRateIndex(1 + 0.25 * Math.pow(2, failure.level
    - 1));
8.              failure.finishTime = Date.now();
9.              failure.status = 'stopped';
10.             break;
11.         case 'smartContract':
12.             clientControl.controlRateIndex(0.5);
13.             failureClient.controlRateIndex(0.5, failure.label);
14.             await failureClient.startFailure(...);
15.             await failureClient.stop(failure.label);
16.             break;
17.         case 'nodeFailure':
18.             await consensus.nodeFailure(...);
19.             break;
20.         case 'network-packageLoss':
21.             await network.packageLoss(...);
22.             break;
23.         case 'network-latency':
24.             await network.networkLatency(...);
25.             break;
26.     }
27. }
```

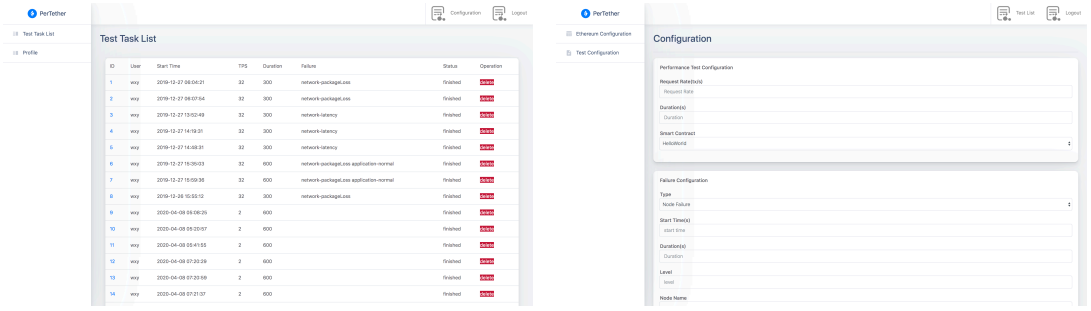
图 4.17: 故障注入子模块-故障注入代码实现

图4.17为故障注入子模块中，failureManager.js 执行故障注入的代码实现。由于故障类型共有四种且可能扩展更多类型，所有类型的故障以 switch-case 的形式区分执行方式。第 17、20、23 行分别代表共识故障中的节点宕机、网络故障中的高丢包率与网络延迟故障。这两类故障注入通过调用第三方 api 实现，代码分别被封装到 consensus.js 与 network.js 中，降低耦合度。共识故障没有参数设置，只有节点宕机和恢复两种状态。网络故障的参数配置为丢包率 30%、60%、90%。第 6-10 行为应用故障实现，流量突增通过线程控制器调整请求速率实现。clientControl 为线程控制器，调用 controlRateIndex 函数控制线程请求速率变化。增长速率的调整幅度为可设置为 1.25、1.5、2 倍请求速率。第 11-16 行为智能合约故障实现，第 12 行调低正常请求的速率为之前的 0.5 倍，第 13 行将低性能合

约的请求从 0 提升到原请求速率的 0.5 倍。正常请求速率调整与应用故障一致，故障请求速率调整由 failureClient 实现。故障的参数配置为智能合约内部的无效循环次数。

4.3　系统示例展示

4.3.1　系统界面截图

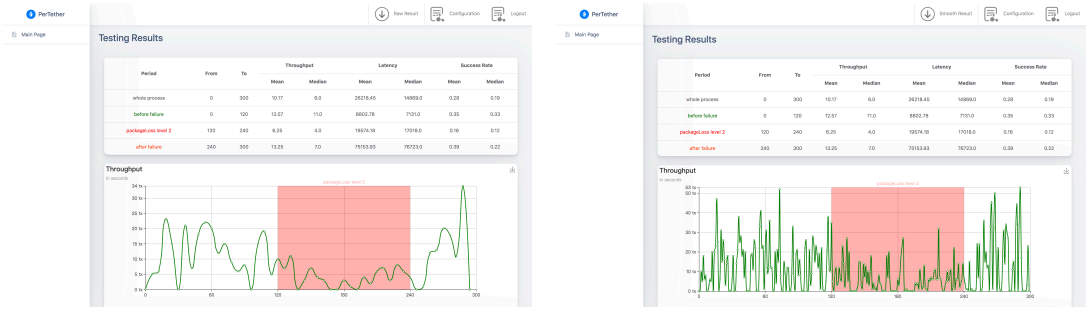


(a) 任务列表　　(b) 测试配置

图 4.18: 任务列表与测试配置界面

如图4.18所示为测试配置与任务界面。左侧导航栏可选择查看任务列表与个人信息，右上方可以跳转至测试配置界面以及登出系统。如图4.18(a)所示，用户完成登录后，主页面展示用户创建和执行的所有测试任务。列表中每一项会展示一些基本的配置信息如请求速率、持续时间等，以及注入的故障类型，还有测试的当前状态，如已完成为 finished，未完成为 pending。点击每个任务的序号可以查看详细测试结果。点击右上角的配置按钮跳转至测试配置界面，如图4.18(b)所示，本界面包含了性能测试配置与故障注入配置，需要用户登录后才可进行配置，用户输入完成区块链配置后跳转至此页面。输入完成且验证通过的配置可以直接点击开始按钮执行任务，任务执行完成后可以前往任务列表页面查看测试结果。

如图4.19所示为测试结果界面。整体测试结果以表格形式展示，表中包含整个测试过程以及故障注入前、中、后的测试指标平均值与中位数。指标的秒级数据将以折线图形式展示。折线图中横坐标为时间，纵坐标为性能指标。注入故障的区间以红色背景突出显示，体现故障对性能指标带来的影响。图中可以看出，故障注入后的一段时间内，系统吞吐量有明显的下降趋势，在故障恢复后迅速反弹。界面中展示了 60% 丢包率的网络故障的故障注入测试结果，图4.19(a)为进行平滑处理后的数据结果，图4.19(b)为未经平滑处理的原始数据结果。平滑数



(a) 平滑结果　　(b) 原数据结果

图 4.19: 测试结果界面

据结构可以较好的展示数据的变化趋势，原始数据中的数据抖动较大，但是数据的最大、最小值会被保留。

4.3.2　测试过程监控

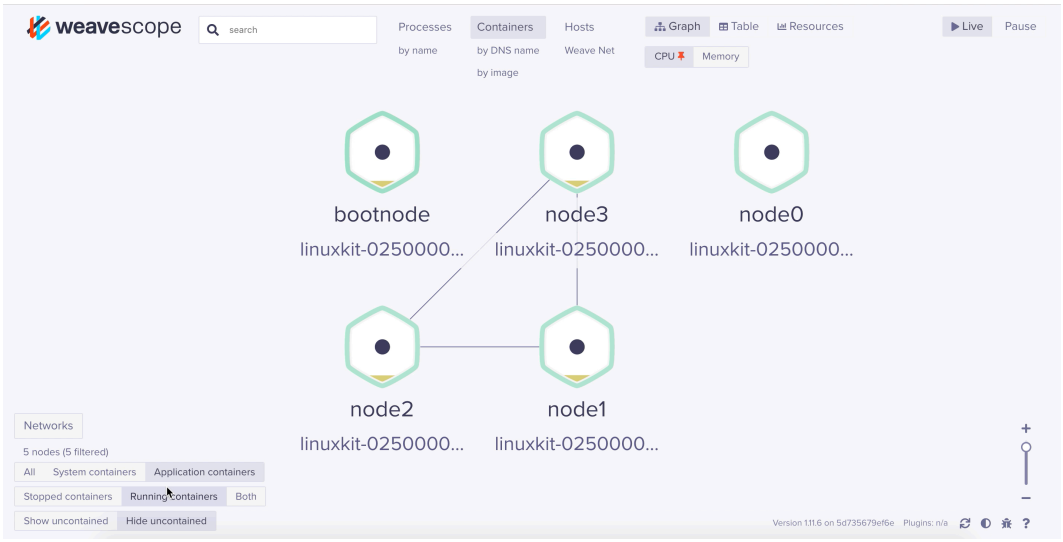


图 4.20: 共识故障时的容器监控

如图4.20所示为共识故障时的容器监控。图中 5 个节点为 Docker 容器，bootnode 用于以太坊节点之间互相识别，剩余四个节点为参与共识的以太坊节点。黑线连接代表他们处于同一个 Docker 局域网网，节点可以通过局域网进行共识。图中为向 node0 注入共识故障后的瞬时监控，node0 容器与其他容器之间的网络连接断开，即退出共识网络。此时，只有 node1、node2、node3 三个节点参与共识。故障恢复时，node0 会重新接入局域网，通过一段时间的节点识别以后重新加入节点共识。

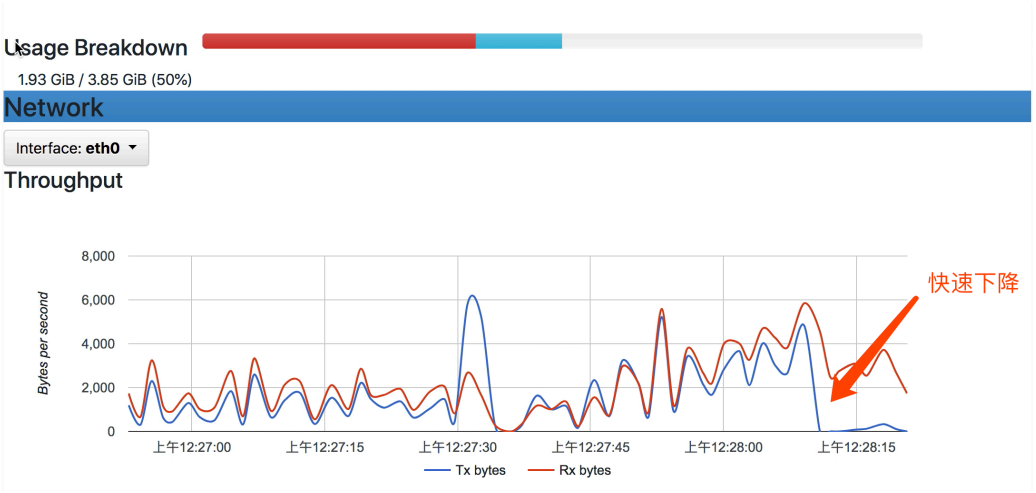


图 4.21: 网络故障时的网络监控

如图4.21所示为网络故障时的网络监控。图中为向 node0 注入网络丢包故障（90% 丢包率）后的 node0 容器的延时网络监控。在故障注入之前，系统以固定速率向节点发送请求，导致节点一直保持一定的网络流量。在故障注入后，可以看出 node0 容器交易数据几乎降为 0。此时，向 node0 发送与 node0 向外发出的数据基本全部丢失。故障恢复时，node0 数据可以正常收发请求。

4.4　本章小结

本章主要介绍了系统的详细设计与实现，从系统框架设计中划分出的两个模块入手，对每个模块进行详细设计。测试管理模块被划分为测试配置子模块与结果管理子模块，测试执行模块被划分为测试准备子模块、性能测试子模块与故障注入子模块。针对每个子模块，通过类图与关键流程顺序图描述其实现与工作流程上的详细设计，每个子模块给出其关键部分代码介绍其实现。最后给出实现完成的系统的展示界面，并给出系统的使用示例。

第五章　实验评估与分析

基于系统的详细设计与实现，本文实现了基于故障注入的以太坊私有链性能测试系统。本章设计并进行了两组实验，用以验证以太坊性能影响因素与故障注入对以太坊私有链性能带来的影响。

5.1　研究问题

通过对以太坊私有链的性能与故障分析，本文提出了两个以太坊私有链性能影响因素，Difficulty 与 Gas Limit，并总结出来四个故障类型，即应用故障、共识故障、智能合约故障、网络故障。为验证性能影响因素与故障对以太坊私有链的性能与稳定性的影响，本节提出两个研究问题：（1）Difficulty 与 Gas Limit 是否会对以太坊的性能产生影响；（2）在以太坊私有链正常运转的过程中，对其注入四种类型的故障后，是否会造成以太坊的性能下降，以及故障恢复后，性能能否恢复。

针对第一个研究问题，本节提出两个研究假设：（1）Difficulty 的上升会导致挖矿所需计算资源上升，进而导致以太坊性能下降；（2）Gas Limit 的降低会导致每个区块容纳的交易数量降低，相同时间内打包的交易数量降低，进而导致以太坊性能下降。

针对第二个研究问题，本节提出四个研究假设：（1）应用故障模拟流量突增，由于以太坊节点无法同时处理大量交易，导致性能下降，并且由于流量增长导致大量交易请求积压，故障恢复后性能无法迅速恢复；（2）共识故障模拟节点宕机，使私有链挖矿能力降低，导致性能下降，并且由于节点重新加入共识网络需要一定时间，故障恢复后性能无法迅速恢复；（3）智能合约故障模拟用户发送大量低性能智能合约请求，导致处理交易的时间延长，进而导致性能下降，并且由于每个共识节点会验证所有打包的交易请求，交易请求会大量积压，性能无法迅速恢复；（4）网络故障模拟网络环境中的高延迟与丢包率，节点收发请求受阻，导致性能降低，并且由于网络故障导致大量交易请求重发，故障恢复时会导致请求量突增，性能无法迅速恢复。

5.2　实验环境

本文实验环境如表5.1所示。由于实现涉及到以太坊测试链搭建，挖矿进程需要较大的 CPU 与内存资源，需要提供性能较好的实验环境。本实验选择 8 核

CPU、16GB 内存的阿里云主机作为以太坊私有链的服务器。以太坊测试链使用 *Docker 1.13.1* 搭建，节点客户端选用以太坊团队官方发布的 Geth 客户端。Geth 客户端有官方发布的 Docker 镜像，本实验选用 *go-ethereum:alltools-v1.8.26* 版本。实验选择 4 个节点构成的以太坊区块链作为测试对象。由于真实场景的以太坊私有链中，可能同时存在挖矿节点与非挖矿节点，本实验中的 4 个节点中 2 个节点为挖矿节点，另 2 个为非挖矿节点。在测试链启动时，首先会启动一个 bootnode，它只占用极低的资源，用于其他以太坊节点互相发现。bootnode 启动完成后才会启动其他节点，所有节点启动在 Docker subnet 中。实验使用 *NodeJS 8.26.0* 启动测试执行模块，智能合约采用 *Solidity 0.4.2* 版本，使用 *web3.js 1.2.2* 与以太坊节点连接，使用 *Pumba 0.7.2* 进行网络故障注入。

表 5.1: 实验环境

参数	**参数详情**
云服务器	阿里云服务器
操作系统	CentOS 7.7 64 位
CPU	8 核
内存	16G
容器	Docker 1.13.1、Docker-compose 1.25.4
以太坊客户端	go-ethereum:alltools-v1.8.26
节点数量	4
挖矿节点数量	2
其他软件	NodeJS 8.16.0、Solidity、0.4.2、web3.js 1.2.2、Pumba 0.7.2

5.3　评价指标

本系统采用吞吐量与交易延迟这两个性能指标来评估以太坊在性能测试、故障注入过程中的性化。吞吐量和交易延迟的计算方法如公式2.1与2.2所示。吞吐量用于评估系统的交易处理能力，即相同时间段内，系统能处理的交易数量越多代表系统性能越好。交易延迟用于评估以太坊针对每个交易处理的速率，即两个相同的交易请求，如果交易返回速度越快，系统的性能越好。

5.4　实验设计

本节设计了两组实验：（1）性能影响因素实验，用以验证两个性能影响因素对以太坊性能的影响；（2）故障注入验证实验，用以验证四类故障对以太坊稳定性的影响。通过实验可以体现这两个方面对以太坊私有链的性能影响。

（1）性能影响因素实验设计

本文提出了两个以太坊中特有的性能影响因素，Difficulty 与 Gas Limit。Difficulty 多以 16 进制表示，最小值为 0x0，代表最低挖矿难度 [41]。Gas Limit 影响一个区块能打包的交易数量，以太坊私有链中 Gas Limit 的默认值为 4712388[58]。在性能影响因素实验设计中，针对两个性能影响因素共设计了两组实验，即 Difficulty 验证实验与 Gas Limit 验证实验。这两组实验根据不同的测试配置各进行 12 次实验，每次实验持续 60 分钟。测试的请求速率从 5tx/s 开始，每间隔 5 分钟递增 5tx/s，到 60tx/s 为止，用以分析以太坊私有链从低请求速率到高请求速率下的性能。

本节实验设计如表5.2所示。对于 Difficulty 验证实验，Difficulty 值设置为从 0x0 开始到 0x160000，每次递增 0x20000，共 12 次实验。本实验中 Gas Limit 设置为以太坊私有链中的默认值 4712388。对于 Gas Limit 验证实验，实验中选取以太坊私有链默认值 4712388 的 0.125 倍至 1.5 倍进行实验，每次递增 0.125 倍，共 12 次实验。为了更好的体现性能的变化趋势，本实验中 Difficulty 设置为最小值 0x0，以保障 Difficulty 对性能的影响降到最低。

性能影响因素验证实验的实验结果通过相关性分析与回归分析进行评估，这两种方法能有效分析出两个性能影响因素与性能指标之间的关系。相关性分析用于研究两个变量间线性关系的程度，本实验采用 Pearson 相关系数，这种相关系数适用于定距连续变量。回归分析用于研究自变量对因变量的影响程度，本实验使用最小二乘法计算回归系数，分析不同请求速率下的性能指标变化趋势。首先，计算每个请求速率下吞吐量与交易延迟的平均值 $m_{throughput}$、$m_{latency}$，得出每个请求速率整体平均值。随后再针对每个请求速率下的指标结果，通过相关性与回归分析得出相应结论。通过计算吞吐量与交易延迟与性能影响因素之间的相关系数 $r_{throughput}$、$r_{latency}$，可以分析出 Difficulty 与 Gas Limit 是否对性能产生影响。通过计算吞吐量与交易延迟与性能影响因素之间的回归系数 $b_{throughput}$、$b_{latency}$，可以得出 Difficulty 与 Gas Limit 是否对性能的影响程度。

表 5.2: 性能影响因素实验设计

实验	参数名称	参数设置					
Difficulty 验证实验	Gas Limit	4712388					
	Difficulty	0x0	0x20000	0x40000	0x60000	0x80000	0xA0000
		0xC0000	0xE0000	0x100000	0x120000	0x140000	0x160000
Gas Limit 验证实验	Gas Limit	589048	1178097	1767145	2356194	2945242	3534291
		4123339	4712388	5301436	5890485	6479533	7068582
	Difficulty	0x0					

（2）故障注入实验设计

如表5.3所示，本节针对实现的四类故障，共设计了 4 组实验，对应应用故障、共识故障、智能合约故障、网络故障。除去共识故障没有参数设置外，其他三种故障均进行了不同参数配置的测试。测试过程请求速率选择 20tx/s，因为根据性能影响因素的实验结果，该请求速率下吞吐量较高且稳定，继续增加请求速率会使系统压力过高导致吞吐量下降。为了数据结构的准确性，测试链启动后会先无请求挖矿 5 分钟，使区块链趋于稳定。每次测试的时长为 480 秒，共分为 4 个时间段：（1）启动时段，第 0-120 秒，前 120 秒由于测试链刚刚开始处理请求，需要进行一段时间预热，此时间段内的指标数据不进行对比；（2）稳定时段，第 120-240 秒，此时间段测试链逐渐趋于稳定状态，可用于与其他时段进行对比；（3）注入时段，第 240-360 秒，第 240 秒时进行故障注入，持续 120 秒，之后故障恢复；（4）恢复阶段，第 360-480 秒，故障恢复后再持续 120 秒正常请求速率用以观察恢复后的测试链状态。这种设计可以让测试链先预热到正常状态，对比稳定状态下的性能指标与故障时段内的指标可以体现出故障注入造成的性能影响，故障恢复后的 120 秒也能体现故障恢复后系统的性能表现。除共识故障外，每种类型的故障会分别使用 3 个参数进行实验，应用故障为流量增长的倍率、智能合约故障为故障负载合约的中循环次数、网络故障为注入的丢包率。

故障注入验证实验的实验结果通过平均值与秩和检验进行评估。平均值拥有对比故障注入不同时段的均值水平，可以直观的看出数据差异，四个阶段的平均值分别由 m_{start}、m_{stable}、$m_{injection}$、$m_{recovery}$ 表示。秩和检验用于对比两组数据之间的差异性，通过计算出 P 值大小是否小于 0.05 判断数据是否存在整体差异。这种方法不受数据的分布方式限制，因此适用于分布不规律的性能指标数据。使用秩和检验可以计算故障注入验证测试中稳定时段（**S**table）、故障时段（**I**njection）、恢复时段（**R**ecovery）相互之间的性能指标数据差异，分别用 P_{SI}、P_{SR}、P_{IR} 表示。通过以上三个值可以得出三个时段内的性能指标是否存在显著差异，进而分析故障对以太坊私有链性能造成的影响。

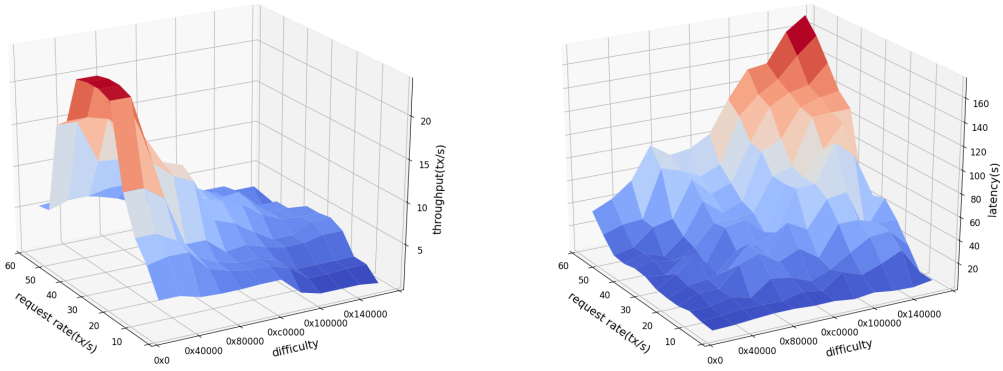
表 5.3: 故障注入实验设计

实验	持续时间	时段				请求速率	故障参数		
		启动	稳定	故障	恢复				
应用	480s	0-120s	120-240s	240-360s	360-480s	20tx/s	1.25	1.5	2
共识							-		
智能合约							10	100	1000
网络							30%	60%	90%

5.5　实验结果与分析

（1）性能影响因素实验结果分析

性能影响因素实验结果通过三维曲面图进行展示。每个图中，x 轴为性能影响因素（Difficulty、Gas Limit），y 轴为性能测试请求速率，z 轴为性能指标（吞吐量、交易延迟）。每个性能影响因素实验结果数数据分析会给出该请求速率下的性能指标平均值 $m_{throughput}$、$m_{latency}$、相关系数 $r_{throughput}$、$r_{latency}$ 与回归系数 $b_{throughput}$、$b_{latency}$，以表格形式展示。。



(a) 吞吐量　　(b) 交易延迟

图 5.1: Difficulty 验证实验结果

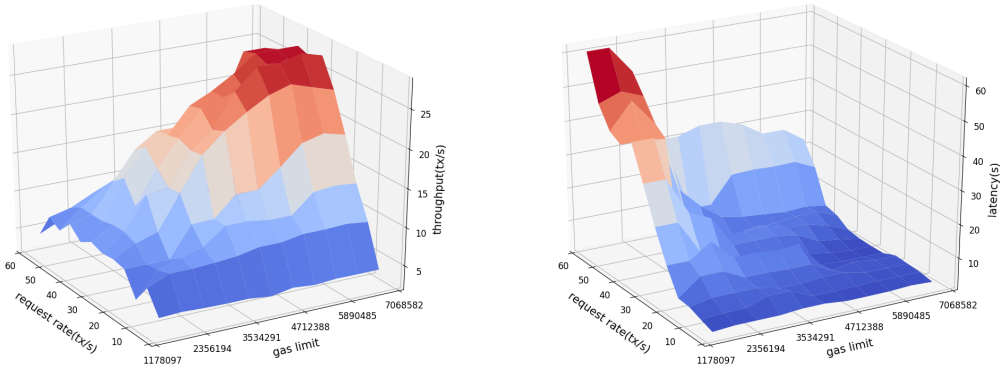
如图5.1为不同 Difficulty 在不同请求速率下的实验结果。图中可以看出，在低请求速率下，以太坊私有链的性能基本稳定，可以正常处理大部分请求。随着交易请求速率的上升，吞吐量首先会达到峰值。Difficulty 越高，吞吐量峰值越低。在吞吐量达到峰值后，请求速率继续增长会导致吞吐量下降，大部分交易请求失败。与此同时，交易延迟随着请求速率增长持续上升。

表5.4为 Difficulty 验证实验结果的数据分析。在吞吐量指标方面，所有请求速率下相关系数绝对值均超过 0.85，即 Difficulty 与吞吐量表现为强相关性。根据回归系数分析，在请求速率为 25-45tx/s 时，回归系数绝对值大于 1.50，Difficulty 对吞吐量的影响较为明显，更高或更低的请求速率下影响不明显。交易延迟指标方面，请求速率 10tx/s 以下时相关系数小于 0.8，Difficulty 与交易延迟的相关性不强，请求速率越高，相关性越强。同时，随着请求速率上升，回归系数大幅上升。由此可以得出，在高请求速率下，Difficulty 对交易延迟的影响更为明显。实验结果表明，Difficulty 确实可以影响以太坊私有链的性能，Difficulty 上升会导致以太坊私有链性能下降，Difficulty 为 0x0 时性能最佳。

表 5.4: Difficulty 验证实验结果数据分析

请求速率	$m_{throughput}$	$r_{throughput}$	$b_{throughput}$	$m_{latency}$	$r_{latency}$	$b_{latency}$
5	2.07	-0.93	-0.38	2.99	-0.30	-0.11
10	3.95	-0.95	-0.63	10.01	0.63	1.34
15	4.44	-0.95	-0.78	16.74	0.81	2.58
20	6.56	-0.86	-0.97	26.72	0.84	3.29
25	10.14	-0.91	-1.84	31.73	0.80	4.48
30	11.34	-0.95	-1.91	41.51	0.83	7.26
35	11.02	-0.93	-1.85	44.40	0.87	8.02
40	10.99	-0.96	-1.91	52.28	0.95	10.87
45	9.30	-0.98	-1.60	58.23	0.96	12.01
50	8.35	-0.93	-1.26	68.32	0.97	12.71
55	5.02	-0.86	-0.39	81.34	0.97	12.85
60	3.56	-0.97	-0.49	96.92	0.97	12.36

如图5.2为不同 Gas Limit 在不同请求速率下的实验结果。Gas Limit 为 589048 时无法部署智能合约，无成功请求返回。在所有请求速率下，吞吐量随着 Gas Limit 升高而升高，在 30tx/s 左右上升幅度最为明显。在 Gas Limit 达到 4712388 后继续增长，吞吐量峰值基本不会继续上升。随着请求速率增长与 Gas Limit 的降低，交易延迟迅速上升。



(a) 吞吐量　　(b) 交易延迟

图 5.2: Gas Limit 验证实验结果

表5.5为 Gas Limit 验证实验结果的数据分析。在请求速率为 5tx/s 时，吞吐量与延迟的相关系数分别为-0.10 与 0.15，Gas Limit 对性能无影响。在吞吐量指标方面，请求速率 55-60tx/s 时，吞吐量与 Gas Limit 的相关系数均为-0.49，对吞

吐量影响较小。其余请求速率下吞吐量与 Gas Limit 呈强相关性。根据回归系数分析，请求速率在 20-45tx/s 时以太坊吞吐量处于较高水平，且随着 Gas Limit 升高变化较明显。交易延迟方面，请求速率 10tx/s 以下时，Gas Limit 与交易延迟的相关系数绝对值小于 0.4，相关性较小。请求速率大于 10tx/s 时，回归系数的绝对值大幅上升，即对交易延迟有明显影响。同时，在越高的请求速率下，交易延迟升高幅度越大。根据三维图与数据分析可以得出，在较低请求速率下，Gas Limit 对以太坊性能没有明显影响。高请求速率下，以太坊性能随着 Gas Limit 上升而提升，但是性能的提升有一定限度，超过一定限度后不会继续上升。

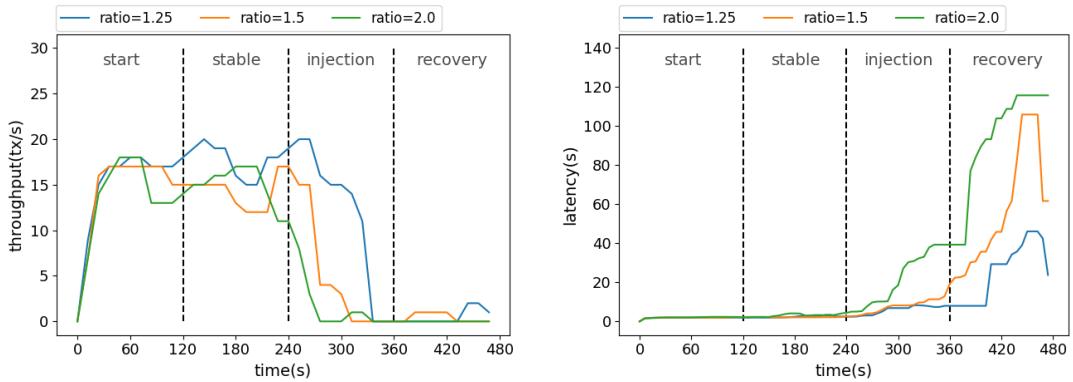
表 5.5: Gas Limit 验证实验结果数据分析

请求速率	$m_{throughput}$	$r_{throughput}$	$b_{throughput}$	$m_{latency}$	$r_{latency}$	$b_{latency}$
5	4.16	-0.10	0.00	2.30	0.15	0.01
10	8.91	0.92	0.24	3.55	-0.35	-0.12
15	10.67	0.93	0.80	4.52	-0.77	-0.23
20	13.48	0.98	1.40	5.99	-0.71	-0.50
25	18.58	0.94	1.95	8.36	-0.88	-1.36
30	20.08	0.97	2.24	8.20	-0.88	-1.90
35	19.52	0.97	2.05	13.30	-0.93	-3.04
40	20.02	0.96	2.09	14.23	-0.77	-3.25
45	18.48	0.97	1.86	16.00	-0.80	-3.62
50	12.85	0.69	0.64	19.37	-0.87	-4.34
55	8.58	-0.49	-0.37	26.55	-0.76	-3.43
60	5.73	-0.49	-0.27	37.64	-0.87	-3.62

（2）故障注入实验结果分析

故障注入实验结果通过折线图进行展示。图中 x 轴为时间，y 轴为性能指标（吞吐量、交易延迟）。实验的四个阶段使用虚线进行划分，不同颜色的线代表不同的故障参数。实验结果中计算每个阶段的性能指标数据的平均值，同时对稳定、注入、恢复阶段的指标数据进行秩和检验，用以判断其差异性。吞吐量与交易延迟的平均值 m_{start}、m_{stable}、$m_{injection}$、$m_{recovery}$ 与秩和检验的 P 值 P_{SI}、P_{SR}、P_{IR} 以表格形式展示。

如图5.3为应用故障实验的性能指标折线图。应用故障注入时分别将请求速率提升为当前的 1.25、1.5、2 倍。在注入应用故障后，随着请求速率的上升，吞吐量没有升高反而下降。同时交易延迟大幅度上升。这种现象证明随着流量增长，短时间内大量的交易对以太坊节点产生较大压力，以太坊的交易请求处理能力降低，而且增长的越快可能导致交易阻塞现象更加严重，处理交易速度越慢。在应用故障恢复后，吞吐量基本无法恢复，仍处于较低水平，而且交易延迟



(a) 吞吐量　　(b) 交易延迟

图 5.3: 应用故障注入实验结果

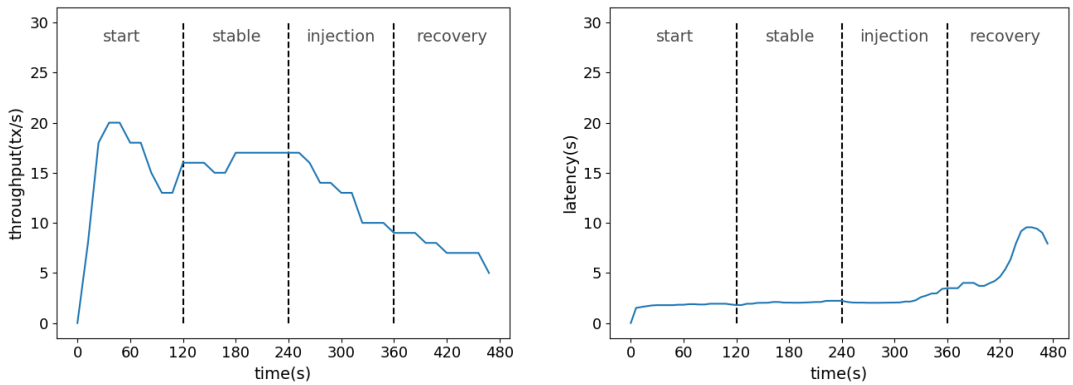
仍然大幅度上升，而且随着请求速率增加，上升幅度更为明显，这说明以太坊在应用故障恢复后不能迅速恢复原状。

表 5.6: 应用故障注入实验结果数据分析

指标名称	增长比率	m_{start}	m_{stable}	$m_{injection}$	$m_{recovery}$	P_{SI}	P_{SR}	P_{IR}
吞吐量（tx/s）	1.25	19.38	19.27	16.50	3.92	0.027	8.9e-17	3.3e-08
	1.5	19.44	19.27	15.34	2.57	4.9e-03	1.2e-15	5.8e-06
	2	19.27	19.20	10.03	1.03	8.5e-06	3.0e-21	3.7e-08
交易延迟（s）	1.25	1.90	2.52	7.38	35.70	1.2e-22	5.1e-41	7.1e-24
	1.5	1.98	2.67	8.58	61.06	9.2e-26	3.2e-41	1.8e-38
	2	2.39	3.44	21.19	93.39	1.5e-33	2.0e-41	1.2e-38

如表5.6为应用故障注入实验结果数据分析结果。根据每个阶段平均值计算结果，故障注入后吞吐量降低，交易延迟升高。同时，三次实验吞吐量和交易延迟的变化幅度有一定差异。流量增长比率越高，平均吞吐量越低，平均交易延迟越高。根据秩和检验结果，P_{SI}、P_{SR}、P_{IR} 均远小于 0.05，即稳定、故障、恢复三个区间的数据呈显著差异。由此可以说明，应用故障对以太坊性能产生显著影响，且故障恢复后性能不能在短时间恢复。

如图5.4为共识故障实验的性能指标折线图。共识故障注入发生在一个挖矿节点。在故障注入执行后，以太坊私有链的交易吞吐量有一定程度的下降，交易延迟无明显变化。算力下降是吞吐量下降的主要原因。当一个挖矿节点退出共识网络，私有链挖矿节点减少，此时系统的算力降低，导致打包区块的能力降低，因此吞吐量下降。由于节点重新加入共识需要一定时间，吞吐量故障恢复后回升继续下降，交易延迟也小幅度上升。



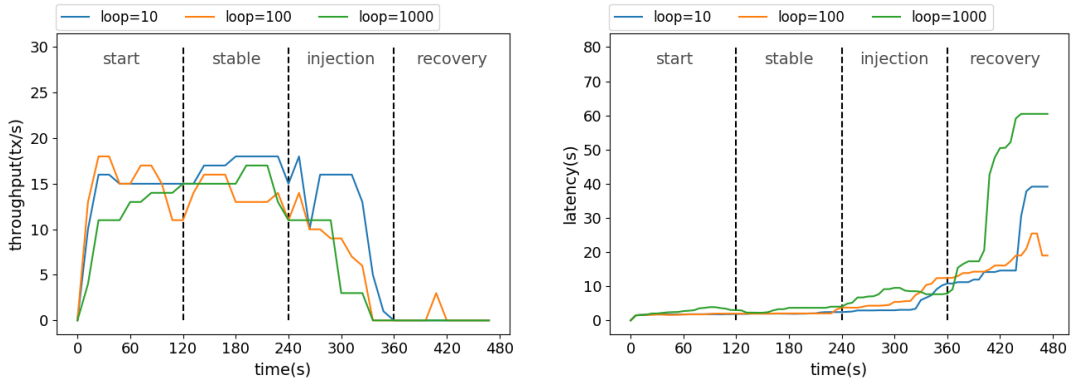
(a) 吞吐量　　(b) 交易延迟

图 5.4: 共识故障注入实验结果

表 5.7: 共识故障注入实验结果数据分析

指标名称	m_{start}	m_{stable}	$m_{injection}$	$m_{recovery}$	P_{S1}	P_{SR}	P_{IR}
吞吐量（tx/s）	19.19	19.46	15.36	10.78	0.024	6.3e-06	4.8e-03
交易延迟（s）	1.83	2.32	2.45	6.88	0.31	6.0e-33	8.8e-30

如表5.7为共识故障注入实验结果数据分析结果。根据三个阶段的平均值，吞吐量均呈小幅度下降。交易延迟的 P_{S1} 为 0.31，可见共识故障注入后对交易延迟无明显影响。根据秩和检验结果，吞吐量在三个时段的 P 值均小于 0.05，因此可以证明故障注入对性能产生了影响。由此可以说明，共识故障对以太坊性能产生影响，但是影响程度有限，故障恢复后性能也不会立即恢复。



(a) 吞吐量　　(b) 交易延迟

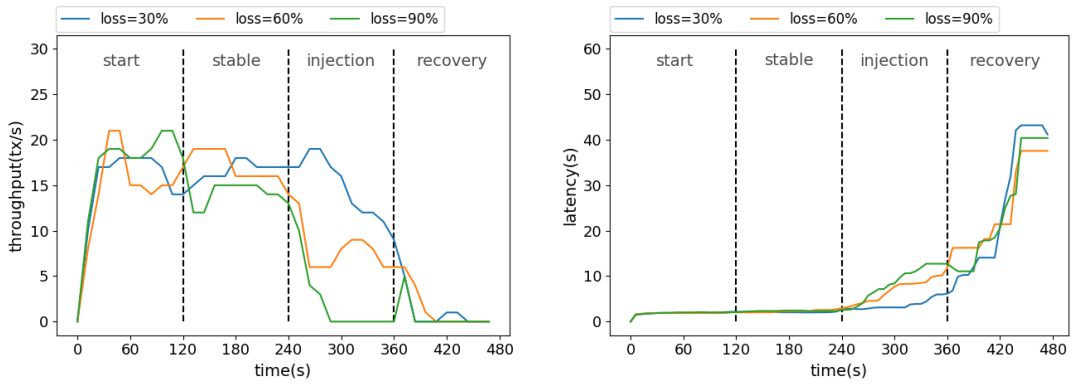
图 5.5: 智能合约故障注入实验结果

如图5.5为智能合约故障实验的性能指标折线图。三次实验的故障参数根据部署的智能合约中的无效循环次数划分，分别为 10、100、1000 次。图中可以看出，在注入故障后，吞吐量和交易延迟均有显著变化。循环次数越多，吞吐量下降速率越快。在故障恢复后，三次实验的吞吐量均维持在较低水平，交易延迟也大幅上升。

表 5.8: 智能合约故障注入实验结果数据分析

指标名称	**循环次数**	m_{start}	m_{stable}	$m_{injection}$	$m_{recovery}$	P_{S1}	P_{SR}	P_{IR}
吞吐量（tx/s）	10	19.29	18.94	17.18	4.21	0.083	1.1e-17	1.2e-11
	100	19.61	19.73	15.91	6.08	3.9e-03	2.0e-11	9.6e-04
	1000	19.23	19.36	13.25	3.32	2.2e-03	1.5e-12	3.7e-05
交易延迟（s）	10	1.83	2.24	4.69	23.50	6.7e-18	3.0e-41	4.3e-35
	100	1.87	2.24	6.33	20.23	1.4e-30	3.5e-41	5.4e-38
	1000	3.12	3.33	8.90	43.53	4.5e-28	5.6e-41	5.9e-30

如表5.8为智能合约故障注入实验结果数据分析结果。根据性能指标的平均值，注入时段的吞吐量相比稳定时段有一定幅度的降低，交易延迟也有不同程度的升高。在故障恢复后，交易延迟升高的幅度较大。根据秩和检验结果，在循环次数为 10 的实验下，吞吐量的 P_{S1} 大于 0.05，故障注入的结果差异不明显，在循环次数为 100 与 000 次时，不同阶段之间的吞吐量与交易延迟均有明显差异。由此可以证明。智能合约故障对以太坊的性能有显著影响，故障恢复后性能也不会立即恢复。



(a) 吞吐量　　(b) 交易延迟

图 5.6: 网络故障注入实验结果

如图5.6为网络故障实验的性能指标折线图。实验在一个挖矿节点的容器执行网络故障注入，三次实验分别为 30%、60%、90% 的丢包率。网络故障对吞吐

量的影响较为明显，越高的丢包率下吞吐量越低。三次实验交易延迟均有上升，且上升幅度基本保持一致。在 90% 的丢包率下，吞吐量迅速下降，区块链基本无法正常处理请求。网络故障在恢复后，吞吐量与交易延迟仍无法恢复。这种现象成因可能是，在故障恢复时丢包率降为 0%，大量请求重发成功，导致节点瞬间收到大量请求，超出了节点能容纳的交易量，导致区块链性能下降。

表 5.9: 网络故障注入实验结果数据分析

指标名称	丢包率	m_{start}	m_{stable}	$m_{injection}$	$m_{recovery}$	P_{S1}	P_{SR}	P_{IR}
吞吐量（tx/s）	30%	19.11	19.40	16.98	5.58	0.081	2.1e-12	3.7e-09
	60%	19.39	19.78	14.09	7.62	5.0e-03	4.1e-11	1.8e-07
	90%	19.62	19.29	9.98	6.22	4.4e-07	2.8e-11	0.081
交易延迟（s）	30%	1.89	2.25	3.86	27.30	7.3e-15	4.8e-41	1.6e-36
	60%	1.98	2.40	8.82	29.44	1.3e-31	3.2e-41	1.2e-28
	90%	1.98	2.40	12.75	31.74	1.6e-30	3.5e-41	1.8e-13

表5.9为网络故障注入实验结果数据分析结果。根据平均值分析，故障注入时段的吞吐量均有一定程度的下降，交易延迟也大幅上升。在故障恢复后，吞吐量处于比故障注入时段更低水平，交易延迟也持续大幅上升。在丢包率 90% 时，由于故障注入时段的吞吐量以降低至极低水平，故障恢复后仍保持在该水平。根据秩和检验结果，在丢包率 30% 时 P 值为 0.081，注入故障前后吞吐量差异不明显。在 60% 与 90% 丢包率下，P 值均远小于 0.05，即稳定、注入、恢复阶段的吞吐量和交易延迟均有显著差异。由此可以认为，网络故障对以太坊性能有显著影响，且这种影响不会随着网络故障恢复而立刻消失。

5.6　效度分析

（1）内部效度分析

本章的两组实验中，每次实验以太坊节点启动后会无压力运行一段时间，待系统稳定后再开始发送请求，这种设计可以最大程度减小由于系统启动时的性能不稳定造成的影响。在故障注入实验中，启动时段作为系统预热，不用作数据对比，最终的结论使用稳定、注入、恢复时段数据，这种设计避免了开始发送请求时可能产生的性能波动对性能指标产生的影响。

（2）结构效度分析

针对性能影响因素验证实验，使用相关性分析作为判断性能影响因素与吞吐量与交易延迟是否有强相关性依据，回归分析作为衡量相关程度的依据。通过计算相关系数与回归系数，能直观的展示出性能影响因素对性能指标是否有影响，依据影响程度的大小。针对故障注入验证实验，首先采用指标的平均值对

不同阶段进行对比，再通过秩和检验判断每段之间是否存在明显的差异。平均值与秩和检验的 P 值能直观反映出故障对性能的影响。

（3）外部效度分析

实验采用 4 节点以太坊私有链进行实验，节点数量可以覆盖小规模私有链的节点数量。在性能影响因素实验中，请求速率以 5tx/s 为间隔递增到 60tx/s 为止，覆盖了从小到大的流量场景。在故障注入实验中，除共识故障外，针对每个故障类型均设计了不同参数，实验设计考虑了不同故障程度下对以太坊私有链的性能影响。

5.7　本章小结

本章主要介绍针对本系统的功能进行的实验，并根据实验结果进行系统评估与分析。根据系统实现的性能影响因素与四类故障，共设计了两组实验进行验证。本章给出了实验环境、两组实验的设计和最终结果。通过分析证明，本文提出的性能影响因素与四类故障确实会影响以太坊私有链的性能与稳定性。

第六章　总结与展望

6.1　总结

区块链技术给人们带来了与传统分布式技术不同的去中心化思想。以太坊的出现将智能合约与区块链技术结合，拓展了区块链的应用场景。用户可以根据需求自行搭建以太坊私有链，但是私有链的性能需要进行性能评估。由于区块链部署场景不稳定，测试环境中模拟真实场景下的故障也尤为重要。

本文以以太坊为基础，结合性能测试与故障注入技术，实现了基于故障注入的以太坊私有链性能测试系统。通过对以太坊进行系统的性能分析，本文得出了两个针对以太坊区块链的性能影响因素 Difficulty 与 Gas Limit。根据对传统分布式系统中的常见故障进行分析，结合区块链、以太坊等独有特性，本文得出四类针对以太坊私有链的故障。在上述分析研究的基础上，最终实现了可以自主变化性能影响因素、可配置故障注入的以太坊性能测试系统。系统设计结合了以太坊技术、Docker 技术、性能测试方法、混沌工程思想，具有一定的创新性。系统实现采用了 Python 和 NodeJS 相结合，分别适应不同模块的不同技术特点，具有一定的开发难度。

根据系统的概要设计，系统最终实现了两个主要模块，即测试管理模块与测试执行模块。测试管理模块实现了测试配置解析管理与测试结果处理展示。该模块实现了完整的前端展示，可以通过浏览器与模块交互，实现配置输入与结果查看。该模块为 Python 实现，Django 服务器为基础，测试配置与结果存储在 MongoDB 数据库中。测试执行模块为系统核心模块，负责测试链搭建、性能测试执行与故障注入。测试链搭建实验 Docker 技术，通过官方镜像搭建 Docker 子网。性能测试执行则启动多线程，通过 NodeJS 的异步支持，批量发送交易请求。故障注入实现为不同故障有相应不同的实现方法，由故障管理器统一管理。

为了验证性能影响因素与四类故障对以太坊私有链的性能影响，本文设计了两种组实验。其中第一组为性能测试影响因素的验证测试，第二组为四类故障注入的验证测试。系统在测试过程中正确运转，能得出清晰、易于理解的测试结果。测试结果表明，性能影响因素与四类故障确实会对以太坊私有链的性能与稳定性产生影响，且不同故障影响的性能指标、程度具有一定差异。由此可见，基于故障注入的以太坊性能测试系统可以达到预期效果。

6.2 展望

目前，本系统已经开源并可以部署使用，但是系统对资源的使用较大，且测试只能在本机进行。后续工作可将系统与测试分离，实现可插拔的测试服务，让测试管理和测试执行分开，简化测试人员的操作。此外，当前区块链部署中可配置项有限，不能覆盖更多场景，后续可以添加启动配置项。

在功能上，系统将继续探究以太坊性能影响因素，研究以太坊私有链上的故障类型，并将其结合到本性能测试系统中，从而扩展系统覆盖的故障场景。随着以太坊的发展，可能出现更新版本的以太坊客户端，以及第三方客户端 Parity，未来系统将兼容更多类型的以太坊客户端。

在实验上，本文中的实验对象为四节点以太坊测试链。在后续工作中，将选用更好性能的云服务器，部署更多数量的节点，并采用更高的请求速率，以探究更高请求压力下的以太坊性能。同时，为了覆盖更复杂的场景，将进行同时多种故障类型注入，探究以太坊在多故障场景下的稳定性与性能。

参考文献

[1] S. Nakamoto, Bitcoin: A peer-to-peer electronic cash system.

[2] R. Wattenhofer, The science of the blockchain, CreateSpace Independent Publishing Platform, 2016.

[3] 袁勇, 王飞跃, 区块链技术发展现状与展望, 自动化学报 42 (4) (2016) 481–494.

[4] 史慧洋, 刘玲, 张玉清, 物链网综述: 区块链在物联网中的应用, 信息安全学报 4 (5).

[5] 刘敖迪, 杜学绘, 王娜, 李少卓, 区块链技术及其在信息安全领域的研究进展, 软件学报 v.29 (07) 270–293.

[6] N. Szabo, Formalizing and securing relationships on public networks, First Monday 2 (9).

[7] V. Buterin, et al., Ethereum white paper: a next generation smart contract & decentralized application platform, First version 53.

[8] Ethereum blockchain explorer., https://etherscan.io/ (2019).

[9] 潘晨, 刘志强, 刘振, 龙宇, 区块链可扩展性研究: 问题与方法, 计算机研究与发展 55 (10) 7–18.

[10] 丁庆洋, 朱建明, 张瑾, 宋彪, 许艳静, 贾传昌, 高政, 基于双层架构的溯源许可链共识机制, 网络与信息安全学报 005 (002) 1–12.

[11] M. Schäffer, M. di Angelo, G. Salzer, Performance and scalability of private Ethereum blockchains, 2019, pp. 103–118.

[12] F. Cristian, Understanding fault-tolerant distributed systems, Communications of the ACM 34 (2) (1991) 56–78.

[13] M. Pilkington, Blockchain technology: principles and applications, Edward Elgar Publishing, 2016.

[14] A. Basiri, N. Behnam, R. Rooij, L. Hochstein, L. Kosewski, J. Reynolds, C. Rosenthal, Chaos engineering, IEEE Software 33 (2016) 1–1.

[15] A. Avritzer, J. Kondek, D. Liu, E. J. Weyuker, Software performance testing based on workload characterization, in: Proceedings of the 3rd international workshop on Software and performance, 2002, pp. 17–24.

[16] 桑圣洪, 胡飞, 性能测试工具 loadrunner 的工作机理及关键技术研究, 科学技术与工程 7 (6) (2007) 1019–1022.

[17] E. H. Halili, Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites, Packt Publishing Ltd, 2008.

[18] Aliyun pts., https://www.aliyun.com/product/pts.

[19] G. Denaro, A. Polini, W. Emmerich, Early performance testing of distributed software applications, in: Proceedings of the 4th international workshop on Software and performance, 2004, pp. 94–103.

[20] T. T. A. Dinh, J. Wang, G. Chen, R. Liu, B. C. Ooi, K.-L. Tan, Blockbench: A framework for analyzing private blockchains, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 1085–1100.

[21] P. Zheng, Z. Zheng, X. Luo, X. Chen, X. Liu, A detailed and real-time performance monitoring framework for blockchain systems, in: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), IEEE, 2018, pp. 134–143.

[22] Blockchain performance benchmarking for hyperledger besu, burrow, fabric, iroha and sawtooth., https://hyperledger.github.io/caliper/ (2018).

[23] M.-C. Hsueh, T. K. Tsai, R. K. Iyer, Fault injection techniques and tools, Computer 30 (4) (1997) 75–82.

[24] H. Ziade, R. A. Ayoubi, R. Velazco, et al., A survey on fault injection techniques, Int. Arab J. Inf. Technol. 1 (2) (2004) 171–186.

[25] W.-L. Kao, R. K. Iyer, Define: A distributed fault injection and monitoring environment, in: Proceedings of IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems, IEEE, 1996, pp. 252–259.

[26] 陈锦富, 卢炎生, 谢晓东, 软件错误注入测试技术研究, 软件学报 20 (6) (2009) 1425–1443.

[27] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, M. Stumm, Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems, in: 11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14), 2014, pp. 249–265.

[28] P. Alvaro, K. Andrus, C. Sanden, C. Rosenthal, A. Basiri, L. Hochstein, Automating failure testing research at internet scale, in: Proceedings of the Seventh ACM Symposium on Cloud Computing, 2016, pp. 17–28.

[29] An easy to use and powerful chaos engineering experiment toolkit., https://github.com/chaosblade-io/chaosblade (2019).

[30] Cryptocurrency market capitalizations., https://coinmarketcap.com/zh/currencies/ethereum/ (2019).

[31] 刘家稷, 杨挺, 汪文勇, 使用双区块链的防伪溯源系统, 信息安全学报 3 (3).

[32] D. Puthal, N. Malik, S. Mohanty, E. Kougianos, G. Das, Everything you wanted to know about the blockchain: Its promise, components, processes, and problems, IEEE Consumer Electronics Magazine 7 (2018) 6–14.

[33] A. Baliga, Understanding blockchain consensus models, Persistent 4 (2017) 1–14.

[34] 袁勇, 倪晓春, 曾帅, 王飞跃, 区块链共识算法的发展现状与展望, 自动化学报 044 (011) 2011–2022.

[35] 刘懿中, 刘建伟, 张宗洋, 徐同阁, 喻辉, 区块链共识机制研究综述, 密码学报 (4).

[36] P. Rogaway, T. Shrimpton, Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance, in: International workshop on fast software encryption, Springer, 2004, pp. 371–388.

[37] A. Baliga, N. Solanki, S. Verekar, A. Pednekar, P. Kamat, S. Chatterjee, Performance characterization of hyperledger fabric, in: 2018 Crypto Valley Conference on Blockchain Technology (CVCBT), IEEE, 2018, pp. 65–74.

[38] M. Vukolić, The quest for scalable blockchain fabric: Proof-of-work vs. bft replication, in: International workshop on open problems in network security, Springer, 2015, pp. 112–125.

[39] A. Kiayias, A. Russell, B. David, R. Oliynykov, Ouroboros: A provably secure proof-of-stake blockchain protocol, in: Annual International Cryptology Conference, Springer, 2017, pp. 357–388.

[40] 贺海武, 延安, 陈泽华, 基于区块链的智能合约技术与应用综述, 计算机研究与发展 55 (11) 112–126.

[41] G. Wood, Ethereum: A secure decentralised generalised transaction ledger, Ethereum project yellow paper (2014) 1–32.

[42] 欧阳丽炜, 王帅, 袁勇, 倪晓春, 王飞跃, 智能合约: 架构及进展, 自动化学报 045 (003) 445–457.

[43] C. Dannen, Introducing Ethereum and Solidity, Vol. 1, Springer, 2017.

[44] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, S. Capkun, On the security and performance of proof of work blockchains, in: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 3–16.

[45] E. J. Weyuker, F. I. Vokolos, Experience with performance testing of software systems: issues, an approach, and case study, IEEE transactions on software engineering 26 (12) (2000) 1147–1156.

[46] 赫建营, 晏海华, 刘超, 金茂忠, 一种有效的 web 性能测试方法及其应用, 计算机应用研究 (1) 281–283+291.

[47] C. Artho, A. Biere, S. Honiden, Enforcer–efficient failure injection, in: International Symposium on Formal Methods, Springer, 2006, pp. 412–427.

[48] 孙峻朝, 王建莹, 杨孝宗, 容错机制测评中的故障注入模型及应用算法, 计算机研究与发展 (11) 56–62.

[49] 胡华平, 金士尧, 王召福, 分布式系统的可信性研究, 计算机工程与科学 (1) 50–55.

[50] O. Kharif, Cryptokitties mania overwhelms ethereum network’ s processing, Bloomberg (4 Dec. 2017).

[51] P. B. Kruchten, The 4+ 1 view model of architecture, IEEE software 12 (6) (1995) 42–50.

[52] 叶俊民, 熊华根, 董威, 齐治昌, 运行时软件故障注入器的设计与实现, 计算机工程 034 (24) 4–6.

[53] A. Holovaty, J. Kaplan-Moss, The definitive guide to Django: Web development done right, Apress, 2009.

[54] K. Chodorow, MongoDB: the definitive guide: powerful and scalable data storage, ” O’Reilly Media, Inc.”, 2013.

[55] T. E. Oliphant, A guide to NumPy, Vol. 1, Trelgol Publishing USA, 2006.

[56] D. Merkel, Docker: lightweight linux containers for consistent development and deployment, Linux journal 2014 (239) (2014) 2.

[57] J. Turnbull, The Docker Book: Containerization is the new virtualization, James Turnbull, 2014.

[58] E. Elrom, Ethereum wallets and smart contracts, in: The Blockchain Developer, 2019, pp. 173–212.

简历与科研成果

基本情况 王新宇，男，汉族，1996 年 8 月出生，黑龙江省哈尔滨市人。

教育背景

2018.9 ~ 2020.6 南京大学软件学院 硕士

2014.9 ~ 2018.7 南京大学软件学院 本科

发明专利

1. 王兴亚，**王新宇**，陈振宇，赵源，孙伟松，“一种基于窥孔优化的以太坊智能合约 Gas 优化方法”，申请号：2019104983796，已受理。
2. 陈振宇，**王新宇**，赵源，王兴亚，何铁科，李玉莹，“一种基于窥孔优化的智能合约性能优化方法”，申请号：2019100557003，已受理。
3. 王兴亚，李紫欣，徐介晖，**王新宇**，陈振宇，“一种基于 LSTM 的以太坊交易打包等待时间的预测方法”，申请号：2019104984233，已受理。
4. 王兴亚，刘芳潇，徐介晖，**王新宇**，陈振宇，“一种基于 Xgboost 的以太坊交易场景下最低燃油价格预测方法”，申请号：2019104983809，已受理。

软件著作权

1. “以太坊性能测试系统”，流水号：2019R11L1328001，已受理。

致　　谢

两年的研究生生涯很快就要过去，在这两年中，我经历了很多，也学到了很多。两年前刚刚本科毕业的我，带着继续充实自己、磨炼自己的态度，开始了研究生的生活。两年后的今天，我深切的体会到，这两年的生活对于我来说有重要的意义。在研究生的生活中，有老师们、同学们、家人们的帮助，让我在知识水平、职业技能乃至社会经验上都有了长足的进步，为我铺好了迈向社会的路上的一块块砖。

首先我要感谢我的导师刘嘉副教授与何铁科讲师。刘嘉老师为人和善、有责任心，对待学术认真严谨，对待学生耐心教导。在我的研究过程中，刘嘉老师悉心指导，为我的学术研究提供极大的帮助。在我遇到困难时，何铁科老师总会给出好的建议，指引我走出困境。在二位老师的带领下，我学到了很多，也收获了很多。

感谢带领我们小组的王兴亚博士后。王老师在面对区块链这样的新领域时，主动带领我们学习新的知识，并在不断学习中发现值得探索的地方。王老师对学术有着一丝不苟的态度，在学术研究上有着丰富的经验，我在王老师身上，学到了对待学术应有的态度。

感谢实验室的领头人陈振宇教授。陈老师平时虽然工作较忙，学生众多，但是对待每一个学生都有一样认真的态度。陈老师每一次开会，都能指出学生的优点与不足，每一次都会指引我进步。

感谢软件学院所有的老师、同学们。感谢老师们的悉心教导，让我的知识、技能有质的飞跃。感谢同学们，人生中有你们的陪伴，使我受益颇多。

感谢一直在我身后支持我的父母、家人们。感谢这么二十多年来对我的养育之恩，有你们的支持才有我今天的成就。如今我即将步入社会，希望能尽我最大努力，报答你们对我的恩情！

最后，感谢所有一直以来关心、照顾我的人！谢谢你们！

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》（以下简称“章程”），愿意将本人的学位论文提交“中国学术期刊（光盘版）电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名：王新宇

2020 年 5 月 26 日

论文题名	基于故障注入的以太坊私有链性能测试系统的设计与实现				
研究生学号	MF1832165	所在院系	软件学院	学位年度	2020
论文级别	☐学术学位硕士 ☑专业学位硕士 ☐学术学位博士 ☐专业学位博士 （请在方框内画钩）				
作者 Email	609402897@qq.com				
导师姓名	刘嘉 何铁科				

论文涉密情况：

☑ 不保密

☐ 保密，保密期（_____年_____月_____日 至 _____年_____月_____日）