青 岛 科 技 大 学

本 科 毕 业 设计

题 目 基于 Raft 一致性算法的

分布式存储系统设计与实现

指导教师 冯**

辅导教师

学生姓名 周**

学生学号 16*******

信息科学技术 **学院** ***** **专业** *** **班**

2020 **年** 06 **月** 01 **日**

基于 Raft 一致性算法的

分布式存储系统设计与实现

摘　要

分布式系统中各节点能否高效、一致的工作是决定分布式系统设计是否优良的重要指标。共识协议是影响分布式系统性能的决定性因素。早在上世纪 90 年代共识协议就已出现，但受制于当时的计算机发展条件，分布式系统并没有得到广泛的应用。近年来，随着区块链技术及其应用的日益火爆，大型分布式系统的一致性共识协议得到越来越广泛的关注[1]。由 Stanford University 的 Diego Ongaro 博士以及 John Ousterhout 教授提出的 Raft 一致性协议因其易于理解，便于工程化实现等特点，在金融、云服务、工业等领域得到了广泛应用，且其稳定性与高效也已得到业界的认可。本文设计并实现了一个基于 Raft 一致性协议的分布式存储系统。该系统的架构和部分流程参考阿里巴巴的 SOFAJRaft，并对系统的日志复制、投票流程以及通信连接等策略进行了优化与调整。本文的主要工作及成果如下：

首先，本文针对业界现有分布式协议的产生背景与使用情况进行调研。主要从拜占庭场景与非拜占庭场景问题，分析现有一致性协议存在的问题，Raft 一致性协议的特点，以及 Raft 的国内外应用现状，以及使用场景等角度开展。其次，从功能性与非功能性两方面分析了分布式一致性存储系统的设计需求问题。根据需求分析结果，对基于 Raft 一致性协议的分布式存储系统的架构和系统的运行流程进行了详细设计。其中，系统架构包括基础服务层、业务层、Raft 服务层以及日志与配置层。运行流程包括节点预选举、选举、日志复制、状态机应用以及线性一致读等。针对现有一致性存储系统方案中存在的日志复制效率低下，通信抖动引起系统故障率较高等问题，本文进行了相应的优化设计。提出节点日志打包，Pipline 日志复制以及通信封装等优化策略。本文提出的基于 Raft 一致性协议的分布式存储系统使用 Java 实现，且全部代码已发布至开源平台 GitHub。

最后，对系统各项功能进行的压力测试以及节点宕机的容错性测试。压力测试的结果表明，本文提出的系统具有可靠性高、容错性较好等优势以及日志应用效率低等劣势；节点宕机容错性测试结果证明，Raft 协议在少部分节点故障的情况下能保证数据可靠性与一致性。

关键词：分布式系统；分布式一致性存储；共识算法；Raft 协议

Design and implementation of distributed storage system based on Raft consensus algorithm

ABSTRACT

Each node works efficiently and consistently or not in a distributed system is an important indicator to determine whether the design of a distributed system is good or not. The decisive factor that can influence this indicator is the consensus protocol used by the distributed system. Consensus protocols have appeared as early as the 1990s, but subject to the development conditions of computers at that time, distributed systems have not been widely used. In recent years, with the increasing popularity of blockchain technology and its applications, the consensus consensus protocol for large distributed systems has received more and more attention. The Raft consensus protocol was proposed by Dr. Diego Ongaro and Professor John Ousterhout of Stanford University. It is famous for its understandable and easy to implement. The Raft consensus protocol has been applied in the fields of finance, cloud services, industry, and other fields. Its stability and efficiency have been recognized by the public. This paper designs and implements a distributed storage system based on Raft consensus protocol. The architecture and part of the process of the system refer to Alibaba's SOFAJRaft, and the log replication, voting process and communication connection strategies of the system have been optimized and adjusted. The main work and results of this article are as follows:

First, this article investigates the background and usage of existing distributed protocols in the industry. Mainly from the perspective of Byzantine and non-Byzantine scenarios, analysis of the problems of the existing consistency agreement, the characteristics of the Raft consistency agreement, and the current status of Raft's domestic and foreign applications, and usage scenarios Secondly, the design requirements of the distributed consistent storage system are analyzed from both functional and non-functional aspects. According to the demand analysis results, the architecture and operating process of the distributed storage system based on the Raft consensus protocol are designed in detail. Among them, the system architecture includes the basic service layer, business layer, Raft service layer, and log and configuration layer. The operation process includes node pre-election, election, log replication, state machine application, and linear consistent reading. Aiming at the problems of low log replication efficiency and high system failure rate caused by communication jitter in the existing consistent storage system solution, this

paper carried out corresponding optimization design. Propose optimization strategies such as node log packaging, Pipline log replication and communication encapsulation. The distributed storage system based on the Raft consensus protocol proposed in this article is implemented in Java, and all the code has been released to the open source platform GitHub.

Finally, stress tests on various functions of the system and fault tolerance tests on leader failures. The results of the stress test show that the system proposed in this paper has the advantages of high reliability, good fault tolerance, etc., and the disadvantages of low log application efficiency. The results of the node downtime fault tolerance test prove that the Raft protocol can guarantee the failure of a small number of nodes. Data reliability and consistency.

KEY WORDS: Distributed system, Distributed consistency storage, Consensus algorithm, Raft protocol

目录

第一章 绪论

本章主要介绍本项目的研究背景及研究意义、国内外应用现状以及论文的主要情况。

1.1 研究背景及意义

在分布式应用场景中，各节点需在网络不稳定，节点宕机甚至恶意节点干扰等问题的情况下保证分布式系统的稳定正常运行。解决此类问题的有效方法是确立并实现一套分布式一致性协议。比特币、以太坊、Hyperledger Fabric 等区块链应用的火爆，使人们认识到了一款高效，稳定且符合应用场景的分布式一致性协议在分布式系统中至关重要。

分布式一致性协议要解决的具体问题是要能保证分布式系统中各节点达成共识。在不可信场景下这个问题就是“拜占庭将军问题”[2]（Byzantine failures）。古罗马时代，东罗马帝国拥有较为广阔的国土。为了守卫边疆，帝国的军队分散在各个地点，相距遥远，军队的将军只能依靠信使来传递消息。若有战争打响，所有的军队必须协同作战才能获得胜利，协同作战的前提是将军们必须达成一致。但是军队中信使可能会有叛徒和间谍，他们在传递消息的时候可能会发出错误的信息。这种错误的信息会扰乱整体的秩序，影响将军们达成共识的同时还可能无法代表大部分将军的意见。这个时候，如何在已知有成员谋反的情况下，剩下未叛变的将军能够继续达成针对战事的一致共识，便是“拜占庭将军问题”（Byzantine failures）。

“拜占庭将军问题”把网络中的节点比做将军，把网络中传输的信号比做信使，是对现实情况的模型化。以比特币为首的公有链是采用工作量证明（PoW）等方式解决此类问题，而以 Hyperledger Fabric 为首的联盟链是采用认证的方式确保节点可信。本文探讨的 Raft 协议属于非拜占庭容错场景，其认定不存在“可能谋反的成员”，但是仍以大多数节点的认可作为达成共识的依据。

在 Raft 共识协议提出之前，学术界已开始采用 Paxos 共识协议来解决分布式共识问题。Paxos 共识协议由 Leslie Lamport 在 1990 年提出，并在 1998 年发表论文《Paxos Made Simple》，但是 Paxos 仍难于被理解，直到 2006 年 Google 使用 Paxos 实现了一个分布式锁服务 Chubby，Paxos 才逐渐进入广大技术人员的视野。

Paxos 协议有一些众所周知的问题：

1) 非常难于理解

2) 版本众多，没有被业界多数承认的工程实现

本文将从探索各类分布式一致性协议开始，逐步引入由 Stanford University 的 Diego Ongaro 博士以及 John Ousterhout 教授的 Raft 共识协议，着重讨论解决上述问题。同时使用 Java 语言从零开始实现一套基于 Raft 的分布式存储系统，并结合业界的大规模应用的

经验，最后进行相应的压力测试。参考项目主要为阿里巴巴蚂蚁金服 RAFT 一致性算法的生产级高性能 Java 实现——SOFAJRaft。

1.2 国内外应用现状

目前流行的分布式一致性协议主要有：Paxos，Raft，Zab，Gossip 等，其分别在不同的项目中得以应用。本小节将介绍现有流行的分布式一致性协议在国内外的应用情况以及其对应项目的特点。

Paxos 应用

Paxos 协议的具体内容将在第二章中提到，此处只简单介绍一下目前的应用情况。Paxos 广泛应用与分布式数据库中，例如 OceanBase。其在 2020 年 5 月的 TPC-C 测试中排行第一。作为使用 Paxos 协议的分布式数据库，OceanBase 具有以下特点[3]：

1) 基于 Paxos 构成的强一致性
2) 集群各节点对等，不存在单点性能短板
3) 高容错性，可容忍少数节点故障

Zab 应用

Zab 协议的全称是 Zookeeper Atomic Broadcast，其借鉴了 Paxos 协议，主要应用于 Zookeeper 服务[4]。Zookeeper 本是 Hadoop 项目的子服务，其为分布式系统提供包括组服务，名称服务，配置维护等功能。主要特点如下：

1) 能够保证顺序一致性，即所有事务请求，最后会被严格的按照顺序应用到 ZooKeeper 中去
2) 能够保证原子性，即整个集群针对某一事务要么全成功，要么全失败
3) 可靠性，即一旦更改被应用，则结果会被持久化。

Gossip 应用

Gossip 协议是 P2P 网络的核心技术，其会随机选择几个临近节点传播消息，而收到该消息的节点也会做同样的操作，直到所有节点都收到了内容[5]。该过程会造成一定的延迟与浪费，但是最终会让所有节点都收到消息。Gossip 协议已应用于 ElasticSearch 的 Node 查找过程，HyperLedger Fabric 的 peer 通信过程中。其具有以下特点：

1) 高扩展性，允许节点随意增加减少
2) 高容错性，任何节点的变动都不会影响 Gossip 协议达成一致
3) 去中心化，不需要有 Leader 节点。该特点也使 Gossip 协议广泛应用与区块链领域。
4) 共识等待时间随时间收敛，即 Gossip 以指数级速度在网络中传播，因此传播速度为 logN，

Raft 应用

据 Raft 的作者统计，截至 2020 年 4 月 30 日，全球共有 114 个 Raft 共识协议的不同实现。此处对应用较广泛，且较成熟的四个项目 etcd、TiKV、braft 和 SOFAJRaft 进行简介。

etcd 项目[6]

etcd 是一款基于 Raft 一致性协议的分布式高可靠的键值 Key-Value 数据库，其由 CoreOS 团队于 2013 年 6 月发起，并使用 Go 语言完整实现。其具有以下特点：

1) 简便：使用配置简便，且可通过 HTTP 协议通信。
2) 高速：根据官方 benchmark，单实例读速度高达 2000 次/秒以上。
3) 安全：支持 SSL。
4) 可靠：使用 Raft 一致性协议，可实现分布式系统的一致性和可用性。

使用场景：

1) 服务发现
2) 消息发布与订阅
3) 分布式锁
4) 高可用性缓存

TiKV

TiKV 是一款分布式事务键值 Key-Value 数据库，其根据 Google Spanner 和 HBase 设计，但可以做到无依赖部署且方便管理，目前已应用于分布式 HTAP 数据库——TiDB。其具有以下特点：

1) 支持跨行 ACID 事务
2) 支持水平伸缩
3) 数据强一致性

braft

braft 是由百度 brpc 团队开发的 Raft 共识协议工业级 C++实现，其在百度内部广泛用于构建高可用的分布式系统。其具有以下特点：

1) 高性能
2) 接口易理解
3) 可搭配 brpc 快速搭建各类分布式系统

使用情况：

1) 已部署应用于百度云块存储
2) 百度强一致性 MySQL

SOFAJRaft

SOFAJRaft 是由阿里巴巴蚂蚁金服团队开源的基于 RAFT 一致性算法的生产级高性能 Java 实现。其在阿里内部已大规模应用，经历过阿里巴巴双十一的考验[7]。主要特点有以下：

1) 高性能低延迟
2) 支持 Batch
3) 流水线复制（Replication pipeline）
4) 租约读（Lease Read）

1.3 论文主要工作

基于以上背景、研究意义以及业内应用情况，本文将使用 Java 从零实现一款基于 Raft 一致性协议的分布式存储系统，并在理解原理的基础上，针对实现过程中出现的各类问题进行相关优化，最终进行基本功能测试与压力测试。由此本论文的工作内容如下：

1) 梳理Raft共识协议的主要流程与使用的技术，详细介绍Raft共识协议的选举过程，包括开始选举的条件、选举流程、投票流程，同时介绍日志复制流程以及线性一致性读等功能。

2) 详细介绍该系统的架构和设计细节。分别介绍基础服务层、业务层、Raft 服务层以及日志与配置层在本论文提出的系统中所起的作用。再从节点预选举、选举、日志复制、状态机应用以及线性一致读等流程中梳理整个系统运行的步骤。

3) 阐述实现过程中所需要的日志复制与打包问题和通信问题。提出如何解决与优化该类问题并介绍节点日志打包，Pipline 日志复制以及通信封装等优化策略。

4) 对本论文提出的系统的基础功能和性能测试日志和数据进行分析和解读。同时根据测试结果分析 Raft 协议的一致性与容错性表现。

1.4 论文组织结构

本文主要包含个章节，每章节主要内容如下：

第一章序论，主要介绍本论文的研究背景及研究意义，国内外的应用情况，包括业界大规模使用的项目特点和使用情况。最后结合问题和需求提出本文的主要目标，并对论文的结构进行相应的阐述。

第二章相关技术陈述，主要介绍设计实现 Raft 分布式存储系统的所需要的技术支持以及理论支持。包括 RPC 服务，持久化服务，高性能队列以及 Paxos 协议与 Raft 协议的介绍与比较等。

第三章需求分析，将从功能性与非功能性两方面阐述本文所提出系统的相关需求，以保证最终能够完成系统稳定性与容错性测试。

第四章系统设计方案，首先将阐述客户端请求流程，再到系统内部架构设计，再从领导人预选举与选举开始，到日志复制，状态机应用与线性一致读方面介绍系统详细设计方案与实现细节，并参考业界实现方案进行分析与讨论

第五章系统优化策略，因 Raft 协议涉及系统往往应用于高性能分布式场景，因此需要保证使用 Raft 的分布式系统具有较高的性能。本论文所提出的系统在实现过程中发现诸多待优化的问题，本章节主要阐述该系统如何从日志复制与储存，投票，通信等方面进行优化。

第六章系统测试及结果分析，本章将使用三台规格相同的 ECS 实例虚机分别运行本论文所提出的系统，并针对响应的功能点进行单元测试与集成测试。完成基本功能测试后将进行压力测试，最后进行结果分析。

第七章总结与展望，本章将进行论文工作总结，分析现有问题与今后工作思路与方向。

1.5 本章小结

本章主要以当前分布式系统应用广泛为前提，引入分布式一致性问题，简要介绍了目前分布式一致性问题的解决方案与解决案例，并阐述了国内外的应用案例。同时制定论文主要工作与组织结构，为后续论文开展奠定基础。

第二章 相关技术基础

2.1 分布式一致性问题及协议

CAP 理论

CAP 理论是指在现有任何分布式系统中，不可能全部达成以下特点：

- 一致性（Consistency），即所有 Peer 在同一时刻都拥有相同的值。
- 可用性（Availability）， 即每次请求都能得到系统的正确回复。
- 分区容错性（Partition tolerance），即可容忍部分 Peer 因网络等原因无法参与服务时分布式系统仍可正常对外提供服务。

根据 CAP 理论，一个分布式系统最多同时满足 CAP 理论三条特点中的两条，而另一条无法完全满足，下面进行分类讨论：

可用性（Availability） & 分区容错性（Partition tolerance）

若只满足可用性与分区容错性，会牺牲一致性特点。因为若有网络分区存在，部分分区中的节点会导致客户端获取的数据不一致。

一致性（Consistency） & 分区容错性（Partition tolerance）

若只满足一致性与分区容错性，则会牺牲可用性特点，因为要保证分区中的部分节点与其他节点一致，则部分客户端不能得到集群的响应。

一致性（Consistency） &可用性（ Availability）

满足一致性与可用性特点的系统往往运行在可靠性与准确性要求较高的场景下。例如金融环境下用户财产存储系统。在此情况下，需要对分区容错性（Partition tolerance）进行处理，Raft 一致性协议则是针对这种场景设计的。

Paxos 协议

Lesile Lamport 于 1990 年提出解决分布式一致性问题的 Paxos 共识协议。在 1990 年的时候，计算机发展还不够发达，因此 Lesile 提出的 Paxos 并没有引起足够的重视，甚至因为它难于理解一时间无法被接受。直到 2006 年 Google 使用 Paxos 实现了一个分布式锁服务 Chubby，Paxos 才逐渐进入学术界的视野[8]。

Lesile 假设有一个名为 Paxos 的小岛，岛上的所有法律（Decress）都需要由信使在议员间传递并由大多数议员表决通过，然后记录到议员的法律簿（Ledger）上后生效。但是由于议员均为兼职，且负责传输消息的信使也是兼职，所以不能保证做决定的时候所有议员在场，或者信使传递消息时一定传递到，且信息有可能被重复传递。因此如何保证即使部分议员不在场，或者信使传递不可靠的情况下，法律仍可以被通过是 Paxos 解决的问题。

本小节将简要介绍 Paxos 算法的各节点角色与运行步骤。

角色介绍

Client：客户端，负责发起请求，不参与分布式系统

Proposer：接收来自客户端的 Request，并向系统提出议案（propose），且负责解决系统中的冲突。

Accpetor：接受议案和投票请求，当议案经过投票后超过半数同意，议案会被接受。

Learner：只读节点，只记录内容，不参与一致性投票。

运行步骤

Phase I：Proposer 收到来自客户端 Request，并提出一个议案，给定编号 A，A 一定要大于之前的编号。

Phase II： Accpetor 接受来自 Proposer 的请求，同时判断如果 A 大于之前接受到的议案，则同意，反之拒绝

Phase III：如果大多数 Accpetor 介绍并通过了 Proposer 的议案，则 Proposer 判断议案通过，并发送 Accept 请求，此请求中含有的内容主要是当前议案的编号和相关议案的数据。

Phase IV：如果在此期间 Acceptor 没有收到任何大于 A 的请求，则接受该请求

由此看来，Paxos 的运行流畅似乎并没有十分复杂，但是实际工程实现起来却较为困难。原因主要是 Paxos 并不是强 Leader 的协议，也就是说在同一时刻可以有多个 Proposer 提交议案，这会使不可控的情况更多且更复杂。事实上 Paxos 难于理解的部分大多根本原因于此。因此需要一款易于理解，且适合工程化应用的一致性协议。

Raft 协议

Raft 协议是由美国 Stanford University 的 Diego Ongaro 博士以及 John Ousterhout 教授提出，其相对 Paxos 算法更易理解，且更容易工程化实现[9]。根据 Ongaro 博士的设计，将一个分布式公示协议分为以下几个部分：

1) Leader Election
2) Log Replication
3) SnapShot
4) Membership Changes

本小节将从以上几个方面介绍 Raft 共识协议。

Leader Election

领导人选举（Leader Election）是决定分布式系统能否稳定正常运行的重要决定因素，且因为 Leader Election 作为工程实现的首要流程，Raft 作者本人与业界主要 Raft 实现都在此做了大量工作[9]。本论文提出的系统也针对 Leader Election 做了大量测试与改进，其中部分改进内容参考自蚂蚁金服 SOFAJRaft。

谈及领导人选举，就不得不来介绍一下 Raft 协议的节点角色分类，Raft 共识协议各节点角色一共分为三种（部分后期实现为四种），分别是：跟从者（Follower）、候选者（Candidate）、领导者（Leader）。部分后期应用实现新增了只读节点（Learner），只负责保存数据，不参与投票。以下将介绍各节点之间的关系和转换流程图：

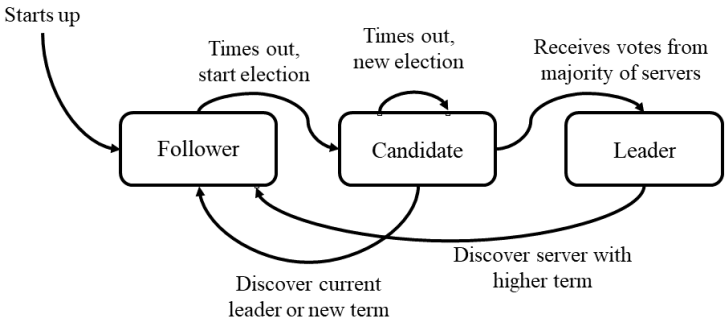

图 1 Raft 身份切换示意图[9]

Figure 1 Schematic diagram of role switching [9]

可以看到，节点启动时，所有节点均为跟随者（Follower）一定时间后若未收到来自领导者（Leader）的心跳（Heartbeat）请求则执行超时操作，将自己的身份转换为参选者（Candidate）并开始选举。选举过程则是首先自己给自己投票，再向其他节点发送投票请求。收到投票请求的节点则会检查该参选者是否满足被投票的[1]条件，如是否拥有比自身更长的 Log Index 等。如果条件满足则会给予该 Candidate 投票。如果一个 Candidate 收到大多数节点的投票，则将自己的身份切换为领导者（Leader）执行后续操作。为了防止所有节点同时超时造成选票被均等瓜分的情况，Raft 协议设计了节点的选举超时时间是一个区间内的随机数，也就保证了所有节点都是随机超时，避免了同时超时瓜分选票的情况。

Log Replication

在 Raft 共识协议中，将达成一次共识的过程抽象为一条日志（Log Entry），而将该日志从领导者节点传输到各跟随者节点并被接受的过程，称为 Raft 协议的日志复制（Log Replication）。本小节将介绍日志复制过程中的日志索引指针（Log Index）和日志轮数（Term）的作用、日志复制的流程以及日志复制的意义。

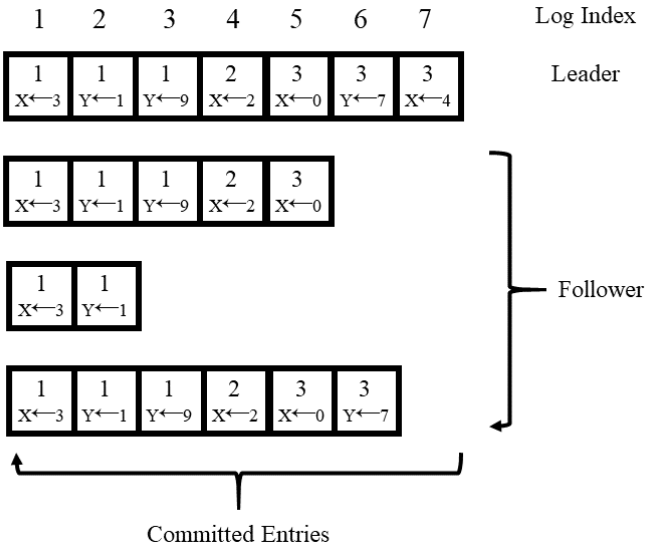

图 2 Raft 协议 Log Index 与 Log Term 示意图

Figure 2 Schematic diagram of the Log Index and Log Term of the Raft protocol[9]

如图 9 所示为使用 Raft 共识协议的系统中日志复制情况、日志索引指针以及日志轮数（Term）情况。每位领导者在上台前都会在之前领导者的日志轮数（Term）上自增一，以此来代表一个新的领导者领导的开始。而日志索引指针（Log Index）则代表日志复制进行的位置，领导者每发出一个 Log Entry 都会在日志索引指针（Log Index）上自增一，当日志复制到大多数节点后，该日志便被认为是已提交 Committed，且不能被更改。

Raft 共识协议的复制流程如下：首先 Leader 节点接受到来自 Client 的请求后，将请求封装成 Log Entry 并储存。储存完成后将日志数据和对应的日志轮数和日志索引指针发送给各个 Follower 节点，各个 Follower 节点接受到 Leader 发来的 AppendEntries 请求后，会验证请求的合法性，并在验证通过后将日志储存并通知 Leader 是否认可该日志。Leader 收到超半数 Follower 节点的同意请求后便可以将该日志进行 Apply，并通知所有节点 Apply 该日志[9]。至此日志复制流程完成。

日志复制过程在强 Leader 措施的保证下，能够降低实现难度，减少需要考虑的情况。各个 Follower 只需要从 Leader 获取 AppendEntries 请求并验证合法性即可，不会产生 Paxos 的死锁情况。而在 Leader 宕机的情况下，日志复制若超过半数节点，则在选举时仍能通过校验参选者的 Log Index 长度来保证选举出来的新 Leader 一定包含之前所有已提交的日志数据[10]。至此，日志复制保证 Raft 共识协议强一致性的意义凸显。

SnapShot

快照（SnapShot）是指将运行 Raft 的分布式系统早期的日志数据进行打包，形成一个 State，即为 Raft 快照。新加入的节点可以通过直接安装快照的方式快速追赶上系统其他节点，同时快照所涉及的日志可以丢弃以节省磁盘空间。

Membership Changes

一般情况下，分布式系统中有成员变更，例如添加或删除节点，需要停止集群并重新启动，这在生产环境下是不能容忍的。因此 Raft 作者设计了一种方案可以动态变更成员，该方案被称为 Joint Consensus，本小节将列举 Membership Changes 的流程、分析该做法能够成功热切换的原因。

以下为 Raft 协议在成员变更过程中所需要经历的三个阶段和两个步骤：

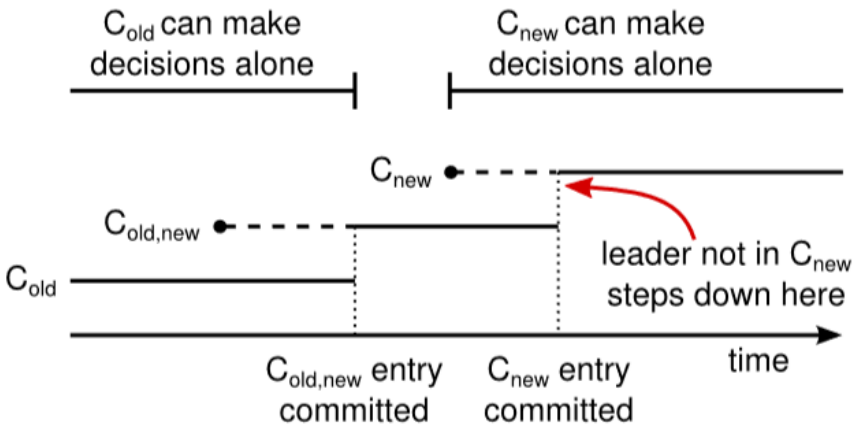

图 3 成员变更示意图

Figure 3 Membership Changes diagram

Phase I：当 Leader 收到更新配置的请求时，Leader 会将新配置与老配置做合并，生成一个 Joint 配置，图中被称为 C_old_new。Leader 将此配置发送给所有 Follower，直到大多数节点已经成功提交该配置。

Phase II：当大多数节点成功提交该配置时，Leader 会将新配置的内容发送给所有 Follower 节点，直到大多数节点已经成功提交该配置。

Raft 协议在完成成员变更过程中需要经历的三个阶段和两个步骤的主要原因如下：

若 C_old_new 共识在达成一致之前 Leader 宕机，则所有现有机器仍可以以旧配置选举新 Leader。

若在 C_old_new 共识达成一致后 Leader 宕机，则所有机器只能在 C_old_new 中选举新 Leader。

若在 C_new 共识达成一致后 Leader 宕机，则所有机器只能在 C_new 中选举新 Leader。

若没有 Joint Consensus，则有可能会出现脑裂情况，即新配置一个 Leader，旧配置一个 Leader。

2.2 分布式一致性存储系统实现技术

2.2.1 RPC 服务与序列化

RPC（Remote Procedure Call）的定义

RPC 是远程过程调用，即在分布式计算中，计算机程序使调用方法的过程在不同的地址上运行（通常是在另一台计算机上）。因其编写方式就如本地普通方法的调用一样，可以使程序员无需明确远程调用过程中的传输与编码细节。

使用 RPC 通常会使用一些常见的 RPC 框架，常见的有 Dubbo，gRPC，SOFARPC 等

序列化的定义

在计算机中，将数据结构或对象转化为可以被存储或传输的二进制字节的过程被称为序列化。反之，将二进制字节转化为可被识别或处理的数据或对象的过程被称为反序列化。不同的计算机语言中，数据，对象以及二进制的表示方法都不尽相同，甚至不同设备，不同操作系统中储存的数据也有区别。而运行分布式服务的节点往往需要在不同设备，甚至不同语言中运行，这就需要一套完整的序列化协议来解决这种问题。

常见的序列化协议有 XML、Json、Protobuf、JDK 原生。各种序列化协议均有各自特点，本文将截取部分广泛应用与 Java 语言的序列化协议进行比较与讨论。以下是一个不同序列化协议之间的对比表。

表格 1 常用序列化协议比较

Table 1 Comparison of common serialization protocols

序列化协议	简要介绍	优点	缺点
Hessian	低依赖，自定义描述的 RPC 协议	对 Java 语言支持较好，性能高	跨语言支持较弱
Kryo	Java 对象图形序列化框架	速度快且序列化后的数据体积较小	跨语言支持十分复杂
Protobuf	中间描述型协议，Google 开源	跨语言支持较好，性能高	可读性较差，需要工具辅助生成代码
JDK 原生	JVM 原生序列化	使用方便	速度慢，有安全问题，且占空间
JSON	简介明了的序列化协议	学习成本较低，通用性较高，跨语言支持较好，可读性强	序列化结果较大，性能较差，且有安全性问题

RPC 服务与序列化在分布式服务中的应用

在一个分布式系统中，各个节点之间需要高效通信，同时还需尽可能减少编码过程中对于通信的操作，因此采用合适的 RPC 服务与序列化协议至关重要。

本项目使用的 RPC 框架为 SOFARPC，由阿里巴巴蚂蚁金服开源开发，是一个性能优良、易于拓展、可用于生产环境的 Java RPC 框架。蚂蚁金服所有业务间的 RPC 调用都是采用 SOFARPC，同时还负责提供其他诸如负载均衡，流量转发，链路数据透传，健康监测等功能。

同时本项目采用 Google Protobuf Buffers 作为序列化协议，其是一种与语言无关，平台无关，且可扩展的序列化结构数据的序列化协议[11]。相比于 JSON，XML 等，Protobuf 不但体积更小，处理效率还更高。

使用 Protobuf 需要预先定义 proto 文件，将需要序列化的数据结构定义在 proto 文件中，然后使用谷歌的 Generator 直接生成对应的序列化使用的代码。

如下，若想通过 Protobuf 传输一个 Person 对象，则需在 proto 文件中预先定义

```
syntax = "proto3";
option java_package = "testproto";
option java_outer_classname = "PersonModel";

message Person {
    int32 id = 1;
    string name = 2;
    string email = 3;
}
```

图 4 Proto 文件定义举例

Figure 4 Proto files definition example

随后通过谷歌提供的代码生成工具读取该 proto 文件，即可生成对应的 Java 类，需要传输时则可构建相应的对象后交由对应的 rpc 框架传输。过程举例如下：

```
PersonModel.Person.Builder builder = PersonModel.Person.newBuilder();
        builder.setId(1);
        builder.setName("zsc");
        builder.setEmail("zsc@mails.qust.edu.com");

        PersonModel.Person person = builder.build();

        rpcService.invoke(person);
```

图 5 Protobuf 使用举例

Figure 5 Protobuf usage examples

该过程使程序员无需关心传输，连接维持，通信异常等问题。极大提升了程序的可控性与可操作性。

2.2.2 RocksDB 数据库

在构建存储系统的过程中，势必要考虑到数据落盘与持久化。在本项目中使用 RocksDB 为数据持久化提供服务。RocksDB 是由 Facebook 开源基于 LevelDB 的 Key-Value 数据库，其对 LevelDB 进行了多种优化。RocksDB 内部使用 LSM tree (log-structured merge-tree)来作为数据存储结构，基于 C++编写[12]。

RocksDB 存储数据时先将数据存储在内存中，达到一定阀值后进行合并落盘操作，且其对磁盘的写操作采用 Append only 的方式，即只进行追加写，改删操作只建立在 Append only 的基础上。此操作极大提升了对磁盘的写效率。

同时，RocksDB 还具有以下的特点：

1) 支持存储任意的二进制数据，无论是由 Protobuf 序列化后的数据还是其他类型的二进制数据。

2) 支持 Colomn Family，可根据 Colomn Family 将多个不相关的数据存储在相同 DB 中。

3) 增加了 Write Ahead Log（WAL）的管理机制，方便管理 WAL 文件。

4) 支持 Batch 的方式批量写入数据。

2.2.3 Disruptor 高性能队列

在本论文构建的 Raft 系统中，如何稳定、高效且安全的发布与处理数据，尤其是多个线程中协同处理数据至关重要。阿里巴巴 SOFAJRaft 为解决该问题，引入了由 LMAX 开发的一个高性能队列——Disruptor[13]。本论文所构建的 Raft 系统同样选用该技术。

目前，在 Apache Storm、Log4j 2 等项目中都采用了 Disruptor 以提升性能。本小节将从以下方面展开介绍：

5) Disruptor 高性能队列与其他 Java 内置队列比较

6) 分析现有加锁和 CAS 方式保证线程安全的队列性能瓶颈

7) 分析 Disruptor 队列高性能的原因

8) Disruptor 在本论文的 Raft 系统中使用场景

Disruptor 高性能队列与其他队列的比较

队列的底层实现一般分为数组、堆和链表，例如 Java 中常用的 ArrayBlockingQueue 底层实现为 ArrayList，LinkedBlockingQueue 底层实现为 LinkedList，以下是 Java 中常用的内置队列比较[14]：

表格 2 常用队列比较

Table 2 Comparison of common queues

队列	是否有界	是否有锁	底层实现
ArrayBlockingQueue	有界	有锁	ArrayList
LinkedBlockingQueue	可选	有锁	LinkedList
ConcurrentLinkedQueue	无界	无锁（CAS）	LinkedList
LinkedTransferQueue	无界	无锁（CAS）	LinkedList
PriorityBlockingQueue	无界	有锁	Heap
DelayQueue	无界	有锁	Heap

可以看到，基于 ArrayList 实现的队列主要通过加锁的方式来保证线程访问的安全，基于 LinkedList 实现的队列保证线程安全的方式主要有两种：加锁和无锁（Compare And Swap 简称 CAS）。加锁方式可以实现有限队列，通常若应用条件要求稳定性较高会使用有锁队列，可以避免内存溢出。

现有队列性能瓶颈

加锁

加锁是最常见的保证线程安全的方式之一。通过对高竞争资源的加锁操作，可确保同一时间点只有一个线程可以操作该资源，从而保证线程安全[15]。以下是一个简单加锁示例：

```
ReenTrantLock lock = new ReenTrantLock();
private void run(){
        lock.lock();
        try{
            //Do something...
        } finally{
            lock.unlock();
    }
}
```

图 6 Java 加锁示例

Figure 6 Java thread lock example

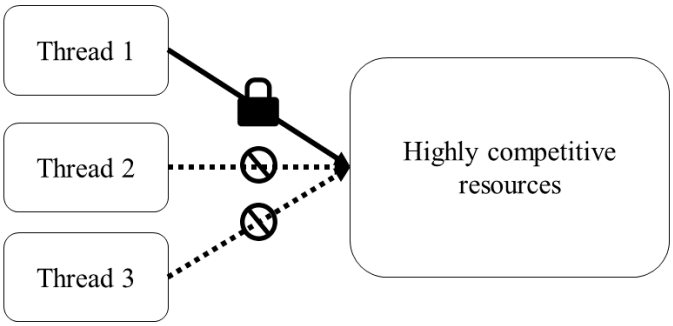

图 7 锁示意图

Figure 7 Thread lock diagram

现实使用过程中，此举会严重影响程序的性能。首先部分锁是悲观锁，即不论资源竞争是否激烈均进行加锁操作，使用完毕后进行解锁操作。此过程在竞争较小的情况下会影响性能。此外，在竞争较大的情况下线程因为获取不到锁而被挂起，在后期获得锁的线程将锁释放后，未获得锁的线程又会尝试恢复[16]。此过程常常伴随着用户态到操作系统内核切换等高消耗操作。

Compare And Swap

Compare And Swap 简称 CAS，是指在某个任务执行过程中，将执行结果与先前进行比较，若成功则交换，若不成功则失败并重新操作，其由 CPU 保证原子性。CAS 会把现在的值和操作之前读出的值进行比较，若相同则把操作后的数值赋给变量，若不相同则进入 while 循环重复执行，直到赋值成功。

CAS 因其是无锁方式实现线程安全，在竞争较小的情况下比加锁有更好的性能，但是在竞争较为激烈的情况下因线程在做无用操作会使性能较差。

伪共享

除上述问题之外，还存在伪共享问题。伪共享是指在 CPU 高速缓存中共享的缓存数据失效导致 CPU 无法充分使用缓存行的情况，此情况会严重影响运行速度。以下是 CPU 与缓存示意图：

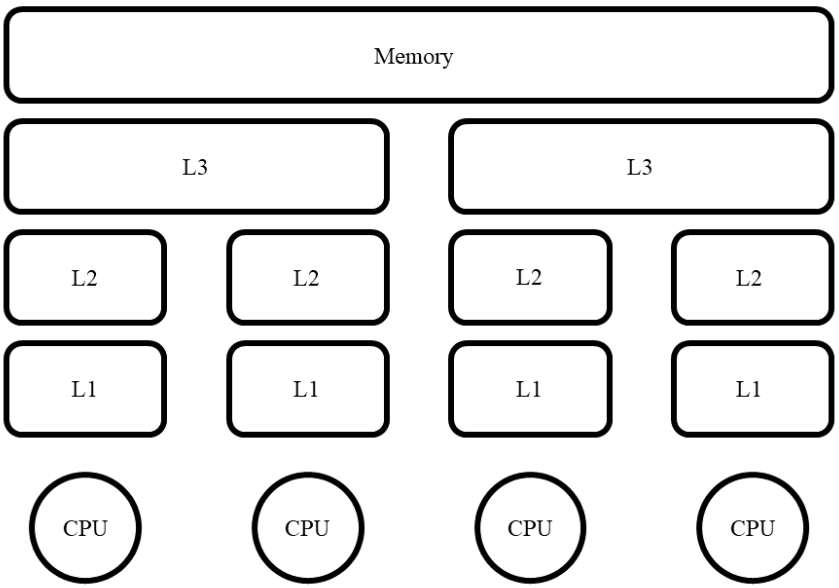

图 8 CPU 内缓存结构

Figure 8 Cache structure in CPU

可以看到 CPU 分为三级缓存，执行运算的时候会先去 L1 获取数据，L1 若查找不到再去 L2，若仍然没有则会前往 L3 查找。最后再到主内存拿。所需要读取的时间是逐渐递增的，最终会成为速度的瓶颈。

CPU 每次从主存中拉取数据时，为加快读取速率，会把要读取数据的周围的数据同样读取回来。但是在特定场景下会使读取到的周围数据失效，这时 CPU 不得不重新拉取数据。若设计不当导致共享的缓存频繁失效，则会导致程序运行效率低下。

在 ArrayBlockingQueue 中有三个 Variable，分别是

1) takeIndex 将要被获取的元素坐标
2) putindex 下一个待添加的元素坐标
3) count 数组中的总元素数量

这三个变量极易被放到同一个缓存行里，且修改其中一个之后，会导致其他的失效，此时 CPU 则需重新拉取缓存。

若要解决伪共享问题，则需要加大可用数据直接的间隔，使得需要用到的数据分别放在不同的缓存行中，以此换得 CPU 的高速访问。实则是以更大的空间占用换取更小时间消耗的举措。

分析 Disruptor 队列高性能的原因

Disruptor 主要采用以下手段来达到高速

1) 采用环形的数组作为存储结构
2) 通过位运算定位元素位置
3) 全过程无锁设计
4) 使用 Sequence 来消除伪共享

2.3 本章小结

本章主要总结了构建本论文的分布式系统所用的技术，包括 RPC 和序列化，Disruptor 高性能队列等技术。同时介绍了 Paxos 共识算法与 Raft 共识算法运行流程和主要组成，为后期构建分布式存储系统时有针对性的考虑优化以及充分实现需求奠定了基础。

第三章 分布式一致性存储系统需求分析

本章从功能性要求与非功能性要求两方面介绍分布式一致性存储系统需要解决的问题，面向的使用场景以及所需要达到的技术指标等内容。

3.1 功能性需求分析

功能性要求主要是为了保证系统在编写后运行过程中各项功能达到预期目标，是系统稳定运行的基础保障[17]。

3.1.1 领导者选举（Leader Election）

能够在系统启动后，在无领导者（Leader）的情况下，经过随机时间超时开启选举流程。且能够最终选举产生有效的领导者。

能够在领导者宕机的情况下，经过超时时间后自动开启领导人选举。

能够避免因为网络分区，与领导者失去联系后节点因为频繁自增日志轮数（Term）导致领导者被强制 stepdown。

3.1.2 日志复制（Log Replication）

领导者能够将日志高速分发至跟随者（Follower）节点，同时能够根据跟随者的日志复制情况重放日志。

领导者日志发送需要以 Batch 的方式发送，提升发送效率。

跟随者接受到日志后需要能够判断日志的合法性。

3.1.3 客户端（Client）

能够以 RPC 服务的方式提供给客户端调用接口。

能够保证客户端无需关心请求节点是否是领导者，就近请求任何节点即可完成请求。

3.1.4 通信服务

能够保证在 RPC 通信的一端无响应、掉线、或超时的情况下，保持一定次数的重试。

能够保证业务 RPC 服务与系统 RPC 服务直接相互隔离。

3.2 非功能性需求分析

3.2.1 独立实现

该系统不会基于任何已有系统开发，但会参考业界主要实现，从零开始开发该系统。完整运行

要求除配置变更（Membership Changes），快照（SnapShot）以外的 Raft 功能要能够完整运行，包括领导人选举，日志复制，FSM 状态机等。

3.2.2 最高运行效率

在系统实现过程中要求编写的系统运行效率最高，即充分考虑高并发，线程安全问题，且在实现后要求进行响应的压力测试。

3.2.3 可容错

要求在系统节点宕机，网络不可达等问题的情况下尽最大可能保持正常运行。在领导者宕机的情况下进行节点选举，在跟随者宕机的情况下不断重试，并支持日志重放以保证节点恢复后尽快跟上日志索引指针。

3.3 本章小结

本章主要从功能性与非功能性两个方面阐明对系统设计的要求，确保设计完成后的系统能够保证 Raft 分布式共识协议的稳定运行。

第四章 基于 Raft 的一致性存储系统设计方案

本章详细介绍本文提出的基于 Raft 一致性协议的分布式存储系统的设计方案。具体包括系统架构和运行流程。首先，设计客户端请求流程，再到系统内部架构设计。随后按照预选举，选举，日志复制，状态机应用与线性一致读的流程完成系统设计与开发。

4.1 系统架构设计

本小节将首先介绍系统节点与节点之间的数据流方向、与客户端的关系开始介绍，并逐渐引入系统内部架构。下图为本论文提出的系统数据流示意图。

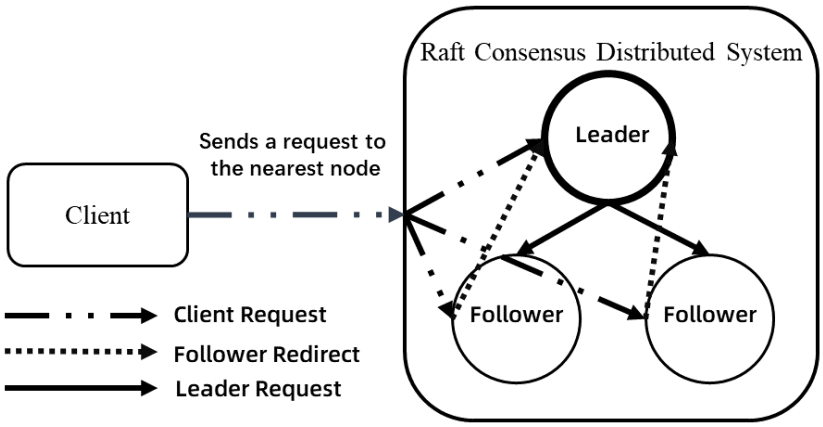

图 9 Raft 分布式共识系统数据流示意图

Figure 9 The data flow schematic diagram of Raft distributed consensus system

客户端（Client）可以选择向就近节点发送请求而无需关心该节点是否是领导者，节点收到请求后会判断自身是否是领导者，若是则执行响应的处理，若不是则会利用跟随者与领导者所构建的业务 RPC 通道转发客户端的请求到领导者。此举有利于节省客户端请求时间，若采用节点判断自身不是领导者后返回给客户端现任领导者地址的方式则会增加客户端请求的流程，且若系统节点采用跨地域部署的方式，客户端与领导者可能需要更长的通信时间来完成请求。因此由就近节点转发请求是效率较高的方式。与此同时可以在图中得出，跟随者除了转发客户端请求外，不能向领导者主动发送消息，这也就保证了数据流只能从领导者流向跟随者，而不能从跟随者流向领导者，从而形成了 Raft 的 Strong Leader 的特性。

除了与客户端的架构图之外，各节点内部的架构图如下。

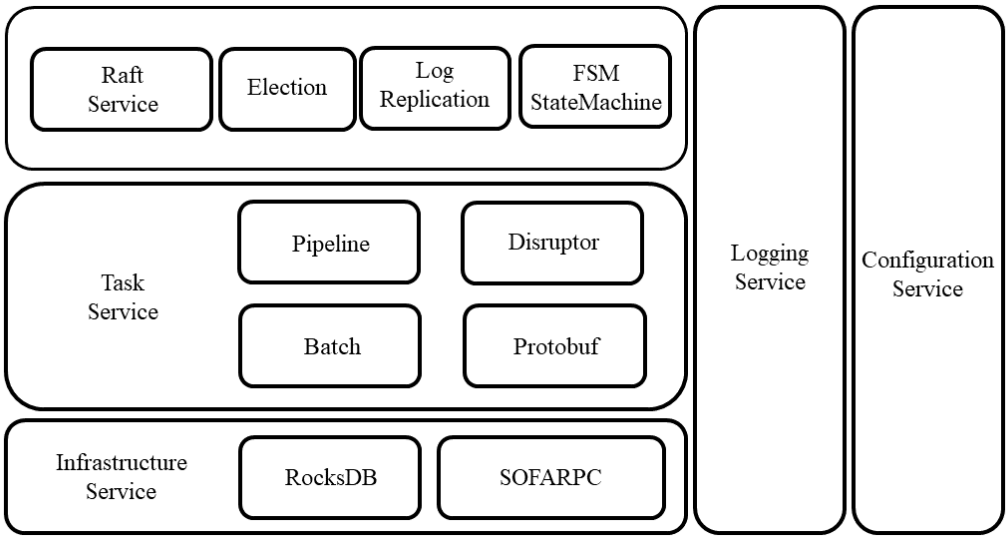

图 10 Raft 共识系统内部结构

Figure 10 Internal structure of Raft consensus system

各节点拥有的功能一致，且仅是身份不同，因此内部架构图均相同，该架构仅限本论文所提出的 Raft 分布式共识系统。由架构图可得，该系统分为底层的基础设施服务层、业务层、Raft 协议层、以及日志层和配置层。各层中使用的技术和框架已在图中列出，针对第三章中的需求，Raft Service 层将独立实现，并与其他层完整结合。接下来将详细介绍各层负责的主要工作和与其他层之间的联系。

首先是基础服务层（Infrastructure Service），该层主要负责基础服务，包括通信与持久化，分别由 RocksDB 与 SOFARPC 负责。在 SOFARPC 内部选用的是 Bolt 通信协议，调用方式（Invoke Type）为回调式（Callback），能够确保所有操作不阻塞，提升系统效率。

其次是业务服务层（Task Service），该层主要负责业务的处理，包括负责协调 Raft 服务（Raft Service）与基础服务的数据传输与批量处理。其中包括日志管道 Pipeline、高性能队列 Disruptor、批量处理 Batch 以及序列化框架 Protobuf。有关 Pipeline 和批处理的相关内容将会在第五章性能优化中详细介绍。

接下来是 Raft 服务（Raft Service），该层主要负责实现与 Raft 共识协议有关的内容，包括选举，日志复制与 FSM 有限应用状态机。有关选举流程，日志复制流程与状态机实现将会在下一小节运行流程中详细介绍。

最后是日志服务与配置服务，主要负责从配置文件中读取相关配置供其余层映射，日志负责记录整个系统的运行情况。

4.2 系统运行流程

本小节将从节点启动开始，介绍领导者选举，日志复制，状态机应用等 Raft 运行流程。

4.2.1 选举流程

各节点启动后，若一段时间内没有收到来自 Leader 的心跳包，则自动进入超时并开启选举流程。若获得超过半数选票，则可以当选领导者，并向其他节点发送心跳包[18]。以下是节点启动时和领导者超时的流程示意图：

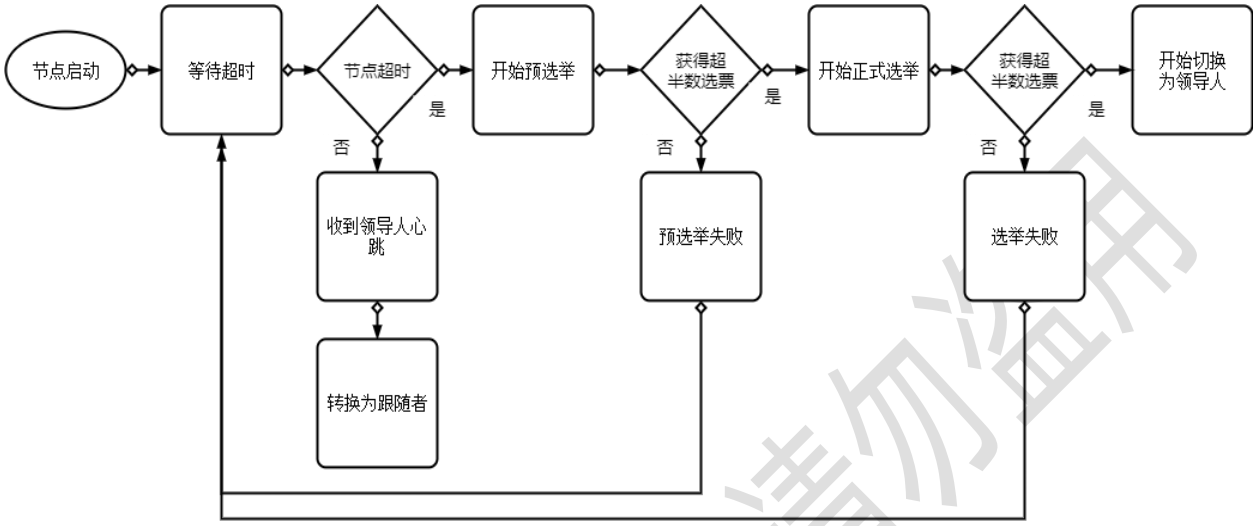

图 11 选举流程示意图

Figure 11 Election process diagram

可以看到，除了正式选举以外还引入了预选举，预选举的目的主要是为了防止部分网络节点恶性自增日志轮数（Log Term）。具体细节如下图所示：

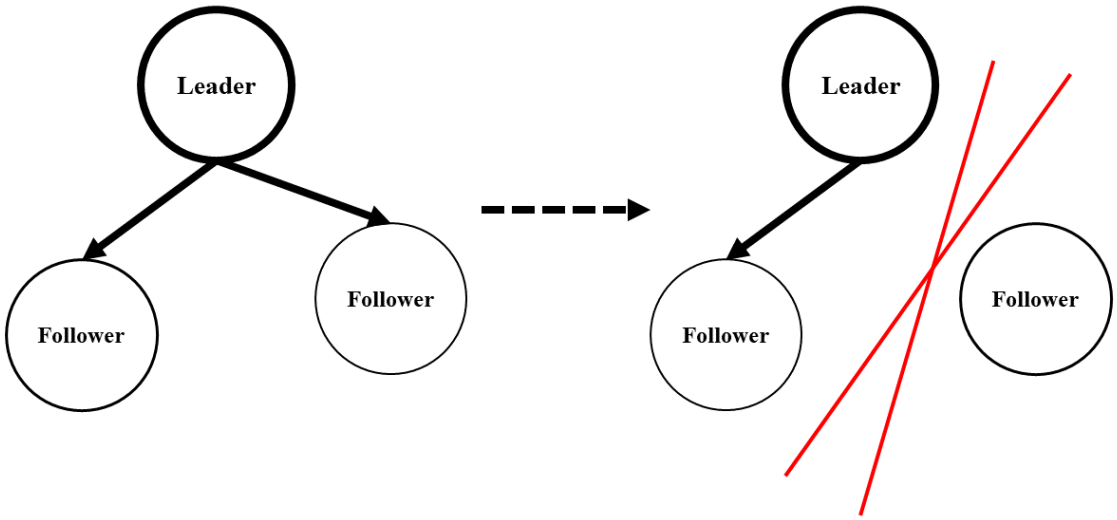

图 12 网络分区导致节点错误选举

Figure 12 Network partition leads to wrong election of nodes

假设系统中有三个节点，一个为 Leader 节点，两个为 Follower 节点。运行一段时间后出现网络波动或网络分区，导致其中一个 Follower 节点出现失去与 Leader 联系的情况，在一段时间收不到来自 Leader 的心跳包时，该节点自动开启选举。

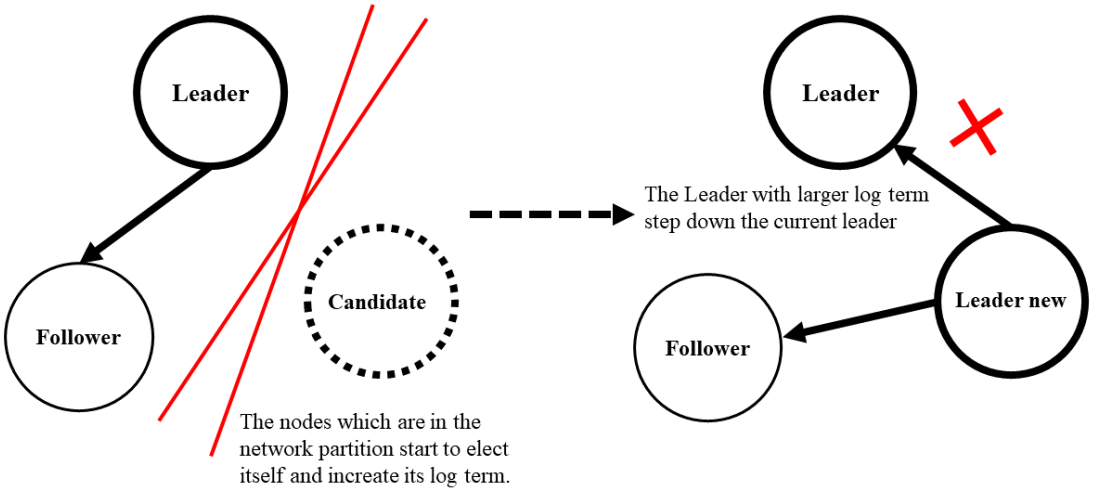

图 13 网络分区导致节点错误选举

Figure 13 Network partition leads to wrong election of nodes

但由于该节点处于网络分区中，并不能获得大多数节点的投票。在频繁开始选举失败后，该节点的日志轮数（Log Term）已被不断自增多次，甚至已经超过了现有领导者的日志轮数。此时若网络分区消失，该节点能够正常访问其他节点，因为其具有较高的日志轮数，所以会迫使当前的领导者 Step down，导致不正确的领导者上任。

该问题的解决方案是通过设置预选举（PreVote），在开启正式选举之前，先进行预选举，确保能够拿到大部分节点的投票后再进行正式选举，此举有效避免了网络分区导致节点不断发起选举的问题。

4.2.2 日志复制与状态机应用流程

完成选举之后，新上任的 Leader 将会向各节点发送 Probe 请求，内容包含了目前所处的日志轮数（Log Term）与日志索引指针（Log Index）。收到该 Probe 请求的节点验证 Leader 的合法性，并完成领导者切换。Probe 请求的本质实际上是空的 AppendEntries 请求。Raft 作者对 AppendEntries 的定义如下：

INVOKED BY LEADER TO REPLICATE LOG ENTRIES; ALSO USED AS HEARTBEAT.

根据定义，AppendEntries 是由 leader 调用，用以做日志复制所用，同时也用于心跳包。一个 AppendEntries 请求的结构如下，业界的各实现会略有不同：

AppendEntries:	
Term	Leader‘s term
leaderId	Leader's ID
prevLogIndex	Index of log entry immediately preceding new ones
prevLogTerm	Term of prevLogIndex entry
entries[]	Log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader’s commitIndex

图 14 AppendEntries 结构

Figure 14 AppendEntries structure

各节点完成领导者切换后便可以等待接受来自领导者的 AppendEntries 请求[19]。领导者收到来自客户端的请求后会进行打包封装，并以 Batch 的方式批量处理日志。从客户端到领导者再到跟随者，这是日志复制的主要流程。具体流程如下：

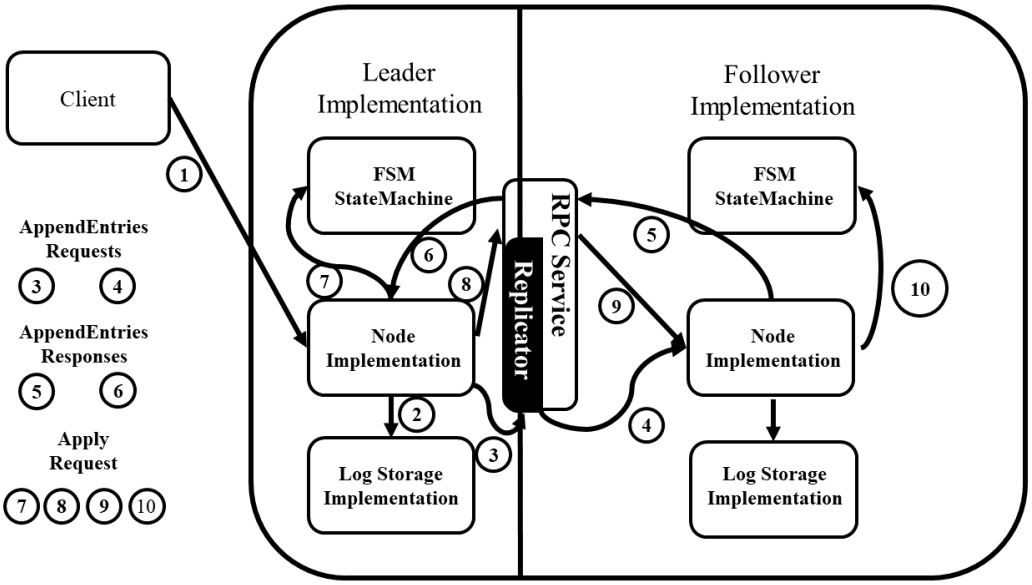

图 15 Raft 共识系统日志复制与状态机应用示意图

Figure 15 Schematic diagram of log replication and state machine application in consensus system

客户端向领导者发起请求（此处已省略向就近节点请求的过程），领导者 Node Implementation 接收请求，并将来自 Client 的请求封装成 Log Entry 并交由 Log Storage Implementation 做持久化操作，完成持久化后将 Log Entry 交由各个 Follower 的 Replicator 发送 AppendEntriesRequest。各 Follower 节点完成日志校验之后，将日志数据做持久化存储，并告知领导者 AppendEntries 结果。领导者根据日志复制情况决定是否将其应用于 FSM 状态机。在应用完状态机后将通知其他 Follower 节点应用状态机。至此日志复制与状态机应用流程完成。

本论文所提出的分布式系统针对日志复制做了相关优化，相关内容请移步第五章 性能优化 查看。

4.2.3 线性一致读流程

接下来是线性一致读。运行 Raft 共识协议的分布式系统能够达到多数节点的一致，其最终目的即是要保证在一个时间点向系统发起请求，在该时间点之后一定能从系统中读取到正确的结果[20]。在单机环境中，线程与线程之间也需要线性一致读，通常都可以通过内存屏障，CPU 原子操作等方式保证线性一致读。以 Java 为例，将每个 Java 程序都看成一个有限状态机，每段 Java 代码都是状态机的指令。在没有进行特殊标注的情况下，JVM 会将指令进行重排序以达到优化的目的。从而导致 Java 程序不能保证运行过程中代码的执行顺序与编写顺序一致，但是可以保证最终结果一致。因此未经特殊标注的 Java 代码不能保证线性一致读。这里提到的特殊标注即是 Java volatile 关键字，该关键字可以保证 JVM 不

对该关键字所标注的内容进行重排序，且所有改变直接刷新至主存，其他线程要获取数据从主内存中获取。使用 volatile 关键字的代码可以做到单机内的线性一致读。但是如果面对多机多节点的情况变无法通过此方式达成线性一致读，因此需要一种协调各节点的方式，保证读出来结果一定是该时间点的正确结果[21]。

在 Raft 分布式系统中有多种方式可以达到线性一致读，因为 Raft 协议本事就是用来达成一致的协议，所以最简单的方法即是所有读请求均走 Raft 协议，在各个节点之间达成一致后读出来的结果一定是正确的。但是由于这种方式会造成不必要的 log 存储和占用大量网络资源，降低写的效率，因此这种方式不是特别推荐。在 Raft 论文中提到两种线性一致读方法，分别是 ReadIndex Read 与 Lease Read。

接下来将介绍两种实现方式 并且在本论文所提出的系统中使用 ReadIndex Read 实现线性一致读。

ReadIndex Read

ReadIndex Read 是相对于走 Raft 协议完成读请求消耗资源更少的实现方式，可以通过 Leader 和 Follower 两处发起 ReadIndex Read。

首先若 Client 请求的是 Leader 节点，则 Leader 节点将自己的当前的 CommitIndex 记录到一个本地变量中。随后开始发起 Heartbeat 请求，若收到超过半数节点返回 Heartbeat Response 则证明自己仍然是 Leader，此时 Leader 只需要等待之前记录的 CommitIndex 应用到状态机之后，便可从状态机中读取数据返回。且该过程无需担心从状态机中读取数据的时候当前节点已不再是 Leader。下图展示的是几种 ReadIndex Read 的情况：

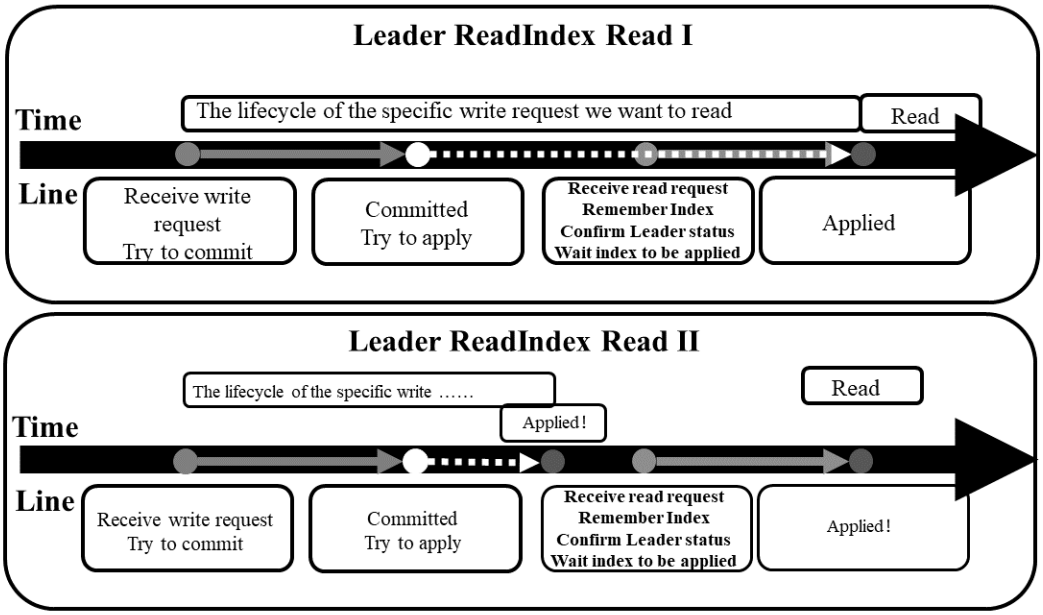

图 16 ReadIndex Read 的几种情况举例

Figure 16 Examples of several situations of ReadIndex Read

可以看到无论读请求发生在什么时候，ReadIndex Read 都能保证读取操作一定是在读请求开始时的 index 应用到状态机之后进行的。这种方式可以保证读请求一定能够读到之前写操作结束后的数据。

同理，若请求的是 Follower 节点，则相对于 Leader 节点多了一个步骤。即 Follower 节点会向 Leader 节点请求最新的 index，leader 节点收到请求后会发送心跳包给各节点确认自己是 leader，并在确认之后告知 Follower 具体的 index，Follower 会等待应用到状态机的 index 到达该 index 后才执行读操作。

ReadIndex Read 支持从 Follower 读取有助于缓解 Leader 节点的磁盘读写压力，提升整体读效率。同时因为该读方式能够在保证线性一致性的同时，省去了日志复制的过程，虽然占用的网络通信，但由于心跳包很小，所以影响较小。

Lease Read

Lease Read 是通过时钟与心跳的方式来实现线性一致读。具体步骤是：Leader 首先发送 Heartbeat 并收到大部分回复之后记录一个时间点，由于 Raft 的选举机制，Follower 节点会在 Election Timeout 的时间后开始重新选举。因此在 Election Timeout 之间，该节点仍为有效 Leader，从而得出该节点的有效时间是 Start Time + Election Timeout ，在此时间之前的请求均可以省掉网络通信，得到快速响应。

Lease Read 因为是与时间挂钩的，因此如果时钟漂移严重，各个节点之间 Clock 走的频率不同，会引起较大的问题[22]。因此在实际设计上，往往要求 Leader 在确定有效时间的时候选择比 Election Timeout 小一个数量级的时间。

第五章 分布式系统性能优化策略

分布式系统在运行过程中，常常会因网络通信波动，磁盘读写压力过大或设计不当等情况导致系统效率低下。因此，需要进行必要的优化。本章介绍了本文提出的系统采用的日志复制优化策略和网络通信优化策略。

5.1 日志复制优化策略

日志的复制流程在之前的章节中已经详细的介绍过了，Raft 的日志复制流程占据整个共识流程的大部分。且因为一条日志对应一个客户端请求，若采用单个 log 单独复制与应用的方式会引起效率低下，严重占用网络通信带宽与磁盘读写[23]。因此适当的优化日志的复制流程对提升共识达成效率至关重要。本小节将介绍本论文所提出的分布式系统中针对日志复制的优化内容，其中部分日志复制优化措施来自 SOFAJRaft。

日志复制主要分为以下几个模块（根据代码实现来分）：节点打包、日志管理（LogManger）、投票箱（BallotBox）。日志复制的优化则主要从这几个模块进行。

5.1.1 节点打包

节点收到来自 Client 的请求后，将首先判断自身是否是 Leader 节点，若不是则将请求进行转发。Leader 节点收到请求后，会进行打包封装，转为可在分布式系统中传递的实体 LogEntry 中，而后进一步封装为 Disruptor 的 Event。该过程若不是在节点压力过大的情况下，不会阻塞。Event 将会被发布至 Disruptor 队列，且该队列会使用 EventHandler 进行处理。不同的是，在 Event 的处理过程中已不再是单个 LogEntry 进行处理，而是以 Batch 的方式进行处理。Batch 的分组方式由 Disruptor 负责，其可保证将多个近似同时到达的 LogEntry 归为同一个 Batch，EventHandler 将可以对此 batch 进行统一操作，从而使用更少的资源服务更多的请求。

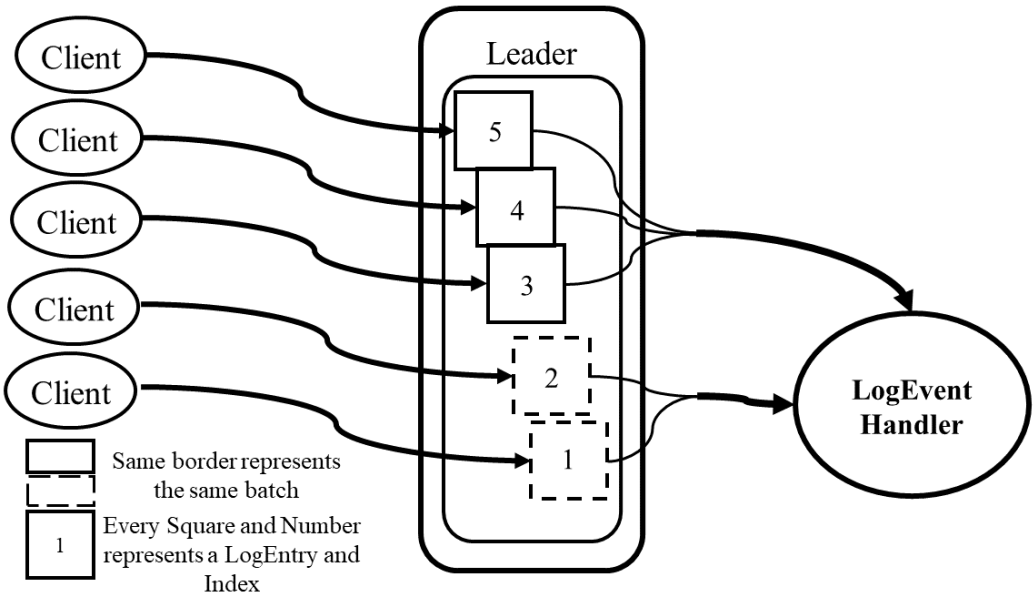

图 17 采用 Batch 方式处理日志示意图

Figure 17 Schematic diagram of processing logs in batch mode

LogEventHandler 收到来自 Disruptor 的 Batch 以后，将 Batch 封装成 LogEntry 链表的形式发送给 LogManager 进行日志落盘。

5.1.2 日志储存

日志在传输到 LogManager 之后将由其负责日志在内存中的储存与持久化储存。在内存中储存时，阿里巴巴 SOFAJRaft 使用了特殊的数据结构——SegmentList。

储存 Log 的数据结构名为 Segment List，其有多个 Segment 组成，每个 Segment 下也由多个 Segment 组成。这样相对于只使用 ArrayList 储存日志有诸多优点。首先是不会引起大量数据移动，在使用 ArrayList 的过程中，如果 remove 元素，则会引起大量的数据移动（底层 ArrayList 是通过 System.arrayCopy()来实现 remove 的），当数据量较大的情况下会对性能造成影响[24]。而使用 Segment List 则可以在添加或删除元素的时候对 Segment 来进行操作，并且移动 Offset 的位置，这样就无需移动后续的元素。其次是能够在保证增删不引起数据迁移的情况下保证对随机地址 O(1)时间复杂度的查找，极其有利于节点对日志的频繁随机访问[25]。

5.1.3 日志复制

日志在复制的过程中难免会有丢失，损坏的情况出现，如何在保证 Raft 复制的日志是连续且有序的是日志复制要解决的重要问题，且解决效率直接影响分布式系统的效率。首先分析场景：大量的日志被并发的发送到 Follower 节点，Follower 节点应该接受所有日志并确定日志的有效性。该过程最简单是实现方式是 "Request -> Response -> Request"交互方式，如左图，这种方式一方面可以保证被复制的日志一定是正确的，另一方面还能及时纠正不正确的日志 Index，例如通知 Leader 下一个日志应该是 12 号，而不是 13 号。但是这

种方式会使日志复制的效率极其低下，Leader 需要等待回复才能继续下一条日志。因此在这里引入日志复制的 Pipline 机制，Leader 仅需要针对每个 Follower 维护一个队列，并记录一下已经发送的日志，如果收到来自 Follower 的回复告知缺失日志，则根据队列进行日志重放。示意图如下：

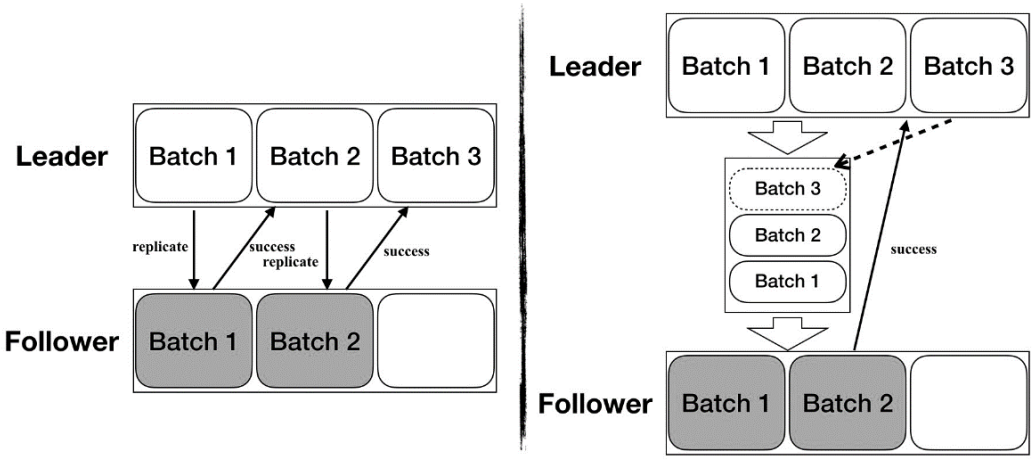

图 18 两种日志复制方式

Figure 18 Two kinds of log replication methods

为此采用每个 Follower 配备一个 Replicator 的方式来进行。成为 Leader 之后，节点将会生成 Replicator Group，为每个 Follower 配备专属的 Replicator，该 Replicator 将全权负责该节点的日志复制工作[26]。示意图如下：

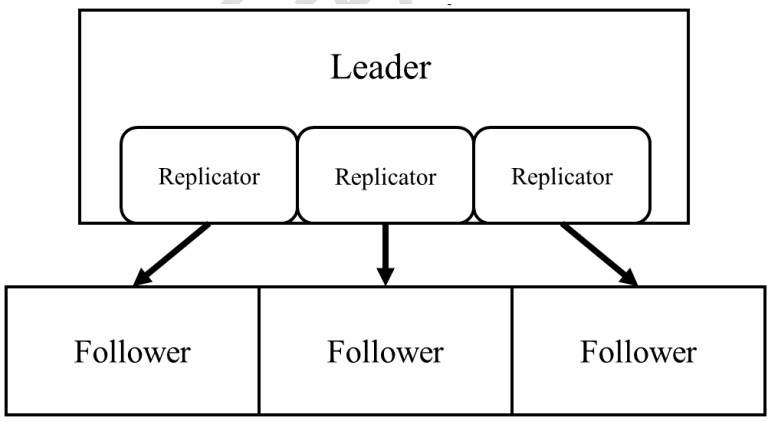

图 19 Raft 系统并发日志复制架构

Figure 19 Raft system concurrent log replication architecture

每个 Replicator 在生成后会自动向 Follower 发送 Prob 请求，其本质就是空的 AppendEntries，由此来获取 Follower 的日志复制情况。Replicator 在开始正常工作后，会将所有发送出去的 LogEntry 封装成 Inflight，并将 Inflight 放入队列中，以便后续在日志复制出现问题的情况下重放日志。

5.1.4 投票箱

Leader 完成日志落盘后，将会新建投票箱（BallotBox），由投票箱判断各个日志是否已被大多数的节点收到并复制。在这里投票箱主要进行以下几项任务：判断需要达成同意的最低节点个数，判断是否执行日志应用到状态机，维护 Inflight。其中判断需要达成同意的最低节点个数仅通过 peerList 节点列表即可判断。而判断是否可以执行日志应用到状态机则有所不同。需要先根据 AppendEntriesResponse 来确定是否投票达到指定人数，再确定日志是否是有序的。如果是有序的，则可以直接应用至 State Machine。若是无序的则标记为“可应用”后等待缺失的日志完成复制并应用后，再来应用此日志。

5.2 通信优化策略

分布式系统中的通信问题

在分布式系统实际运行过程中会遇到网络不可用，时延抖动等情况出现，出现此类情况下则需要 RPC 服务层做相应的重试[27]。但是实际开发的过程中发现若点对点连接出现故障会导致当前线程异常退出，即 FailOver 模式。此时若不采用注册中心的方式则会导致无法继续连接上该节点，但是使用注册中心会在一定程度上降低容错性，节点维持需要靠与注册中心联系。

为进一步提升系统健壮性，避免因为网络抖动等原因致使 RPC 连接断开后线程异常退出，因此需要优化通信层的设计。

通信优化策略

针对上述问题，本论文提出的基于 Raft 的分布式一致性系统针对 RPC 服务层进行了优化设计。在通信层使用了阿里巴巴蚂蚁金服 SOFARPC 框架，并未使用注册中心，仅使用了点对点连接的方式。所采取的通信优化策略如下图所示：

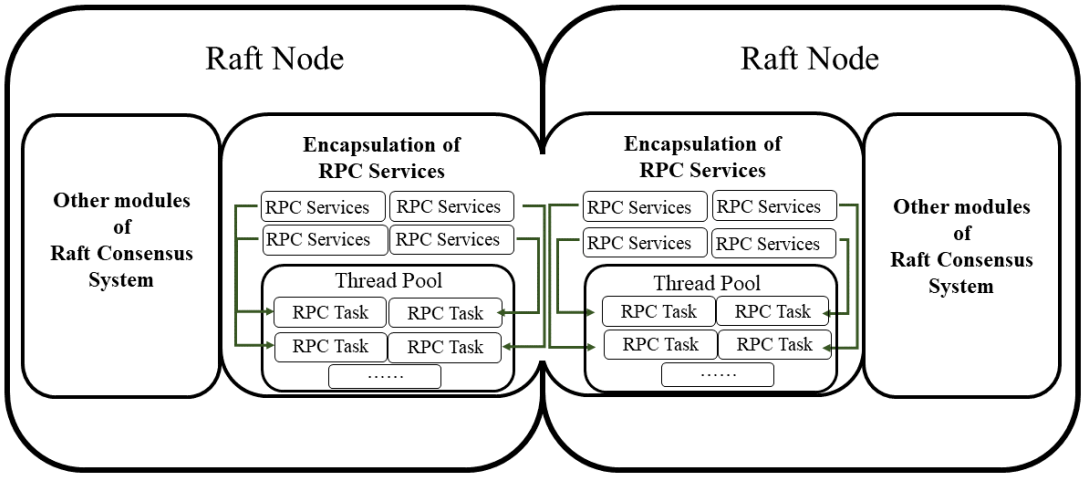

图 18 RPC 服务封装架构

Figure 20 RPC service encapsulation architecture

在 Raft 系统启动时，分别建立各节点的 RPC 服务代理类，即图中的 RPC Services，所有 Raft 其他模块若想调用 rpc 服务均与 RPC 服务封装进行调用。其他模块只需要将需要

通信的内容交给 RPC 服务封装，其他的失败重试与 FailOver 均无需担心，RPC 服务封装便会做好一切工作。这样降低了业务层的复杂度，增强了系统的健壮性。对于 RPC 服务封装，其会根据上层 RPC 服务所交代的数据，将 RPC Service 与数据封装成 RPC Task 交由线程池运行，运行结果通过 Callable 来获取，若任务失败则根据情况自动重试[28]。因为任务是交由线程池运行，所以即使失败也不会影响核心线程。

第六章 系统测试及结果分析

本章针对 Raft 分布式系统中领导人选举、日志复制、状态机应用以及线性一致性读进行基本功能测试，Benchmark 以及节点宕机与网络分区测试。主要目的为测试该系统能够满足 Raft 协议所包含的各种功能，且能否在部分节点宕机的情况下保证系统的稳定运行。本论文所提出的系统源代码将开源至 GitHub 平台，地址详见附件。运行测试用例的环境如下：

表格 3 测试环境

Table 3 The environment of testing

节点	vCPU	内存	处理器主频/睿频	内网带宽	内网收发包	是否同一地域	储存空间	IOPS	操作系统
节点 1	2 vCPU	8 GiB	2.5 GHz /3.2 GHz	1 Gbps	30 万 PPS	是	40GiB	2120	CentOS 7.4
节点 2	2 vCPU	8 GiB	2.5 GHz /3.2 GHz	1 Gbps	30 万 PPS	是	40GiB	2120	CentOS 7.4
节点 3	2 vCPU	8 GiB	2.5 GHz /3.2 GHz	1 Gbps	30 万 PPS	是	40GiB	2120	CentOS 7.4

6.1 领导人选举功能测试结果与分析

本小节将测试Raft分布式系统中领导人选举的相关内容，将从基本选举功能开始测试，再到节点宕机后的重新选举功能测试。

基本功能测试

预选举

各节点启动后，将首先检测到超时，并开始预选举，如下图所示：

```
"2020-05-21 13:25:20 上午 [Thread: Thread-1][ Class:core.RocksDBStorageImpl Method:
core.RocksDBStorageImpl.init(RocksDBStorageImpl.java:75) ]
INFO:Start to init RocksDBStorageImpl on logPath E:\\Rocksdb1\log2
"2020-05-21 13:25:20 上午 [Thread: Thread-1][ Class:core.FSMCallerImpl Method:
core.FSMCallerImpl.init(FSMCallerImpl.java:68) ]
INFO:Starts FSMCaller successfully.
"2020-05-21 13:25:20 上午 [Thread: Thread-1][ Class:core.NodeImpl Method:
core.NodeImpl.init(NodeImpl.java:312) ]
INFO:Node init finished successfully
"2020-05-21 13:25:20 上午 [Thread: Thread-1][ Class:core.NodeImpl Method:
core.NodeImpl.setChecker(NodeImpl.java:336) ]
DEBUG:SetChecker
"2020-05-21 13:25:20 上午 [Thread: Thread-1][ Class:rpc.EnClosureRpcRequest Method:
rpc.EnClosureRpcRequest.<init>(EnClosureRpcRequest.java:35) ]
INFO:EnClosureRpcRequest init succeed with rpcServicesMap size:2
"2020-05-21 13:25:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-1][ Class:core.NodeImpl Method:
core.NodeImpl.lambda$init$0(NodeImpl.java:289) ]
ERROR:Node timeout, start to launch TimeOut actions as it has not received heartBeat for 397701571 ms
"2020-05-21 13:25:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-1][ Class:service.ElectionService
Method: service.ElectionService.checkToStartPreVote(ElectionService.java:22) ]
DEBUG:checkToStartPreVote:true
"2020-05-21 13:25:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-1][ Class:core.NodeImpl Method:
core.NodeImpl.setNodeState(NodeImpl.java:1121) ]
DEBUG:Set node state from follower to preCandidate
```

图 19 预选举测试日志

Figure 21 PreVote test log

随后将向各节点发送预投票请求

```
"2020-05-21 13:25:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-1][ Class:rpc.RpcServicesImpl
Method: service.ElectionServiceImpl.startPrevote(ElectionServiceImpl.java:67) ]
INFO:Send preVote request from raft-2 to PeerId{endpoint=localhost:12200, checksum=0} at 397701598 on term 0
"2020-05-21 13:25:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-1][ Class:rpc.RpcServicesImpl
Method: service.ElectionServiceImpl.startPrevote(ElectionServiceImpl.java:67) ]
INFO:Send preVote request from raft-3 to PeerId{endpoint=localhost:12230, checksum=0} at 397701601 on term 0
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handlePrevoteResponse(ElectionServiceImpl.java:118) ]
INFO:Handle preVote response from: raft-3 result: true
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handlePrevoteResponse(ElectionServiceImpl.java:125) ]
DEBUG:Prevote grant list [raft-1, raft-3]
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:service.ElectionService Method:
service.ElectionService.checkToStartElection(ElectionService.java:30) ]
DEBUG:checkToStartElection:true
```

图 20 预选举测试日志

Figure 22 PreVote test log

可以看到，该节点分别向 raft-1 核 raft-3 节点发起预选举请求，并获得了 raft-1 节点的同意。因为在发起预投票之前已完成自身的投票，所以目前已经得到了三节点中大多数节点的同意，并可以开始选举。

选举

获得多数节点预选举投票的预选举者（preCandidate）可以发起正式选举，如下图

```
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:81) ]
INFO:Current node start election
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:core.NodeImpl Method:
core.NodeImpl.setNodeState(NodeImpl.java:1121) ]
DEBUG:Set node state from preCandidate to candidate
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:104) ]
INFO:Send vote request to PeerId{endpoint=localhost:12200, checksum=0} at 397703070 on newTerm 1
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T7][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:104) ]
INFO:Send vote request to PeerId{endpoint=localhost:12230, checksum=0} at 397703072 on newTerm 1
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T9][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handleElectionResponse(ElectionServiceImpl.java:140) ]
INFO:Handle vote response from: raft-3 result: true
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T9][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handleElectionResponse(ElectionServiceImpl.java:140) ]
INFO:Handle vote response from: raft-1 result: true
"2020-05-21 13:25:20 上午 [Thread: SOFA-RPC-CB-8-T9][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handleElectionResponse(ElectionServiceImpl.java:148) ]
INFO:Current node start to perform as leader at term 1,grant peer list [raft-1, raft-3]
```

图 21 选举测试日志

Figure 23 Election test log

raft-2 向 raft-1 和 raft-3 发起投票请求，并获得了 raft-1 的同意。同样，在发起选举前，raft-2 已经进行自选举，因此已经获得了三个节点中的两个节点的同意。此时 raft-2 已可以开始担任领导者的角色。并开始向其他节点发送空的 AppendEntries 请求。至此，节点选举完成。

节点宕机测试

本小节将测试若正常运行过程中，领导人宕机后其他节点能否正常选举新的领导人。首先正常启动三节点，并等待节点完成初始化与选举流程，而后通过 kill 的方式将领导者节点终止，模拟宕机的情况。观察其他节点能否正常执行选举流程。

```
"2020-05-21 13:26:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-2][ Class:core.NodeImpl Method:
core.NodeImpl.lambda$init$0(NodeImpl.java:289) ]
ERROR:Node timeout, start to launch TimeOut actions as it has not received heartBeat for 1001 ms
"2020-05-21 13:26:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-2][ Class:service.ElectionService
Method: service.ElectionService.checkToStartPreVote(ElectionService.java:22) ]
DEBUG:checkToStartPreVote:true
"2020-05-21 13:26:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-2][ Class:core.NodeImpl Method:
core.NodeImpl.setNodeState(NodeImpl.java:1121) ]
DEBUG:Set node state from follower to preCandidate
"2020-05-21 13:26:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-2][ Class:rpc.RpcServicesImpl
Method: service.ElectionServiceImpl.startPrevote(ElectionServiceImpl.java:67) ]
INFO:Send preVote request from raft-1 to PeerId{endpoint=localhost:12220, checksum=0} at 400066827 on term 1
"2020-05-21 13:26:20 上午 [Thread: JRaft-Node-ScheduleThreadPool--21-thread-2][ Class:rpc.RpcServicesImpl
Method: service.ElectionServiceImpl.startPrevote(ElectionServiceImpl.java:67) ]
INFO:Send preVote request from raft-1 to PeerId{endpoint=localhost:12230, checksum=0} at 400066830 on term 1
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handlePrevoteResponse(ElectionServiceImpl.java:118) ]
INFO:Handle preVote response from: raft-3 result: true
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handlePrevoteResponse(ElectionServiceImpl.java:125) ]
DEBUG:Prevote grant list [raft-1, raft-3]
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:service.ElectionService Method:
service.ElectionService.checkToStartElection(ElectionService.java:30) ]
DEBUG:checkToStartElection:true
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:81) ]
INFO:Current node start election
```

图 22 节点宕机测试日志

Figure 24 Node failure test log

当 leader 节点宕机后，其余节点会感知到超时，并开始随机时间内的预选举操作。随机时间开始选举主要是为了防止所有节点同时开启选举造成选票瓜分的情况。

```
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:core.NodeImpl Method:
core.NodeImpl.setNodeState(NodeImpl.java:1121) ]
DEBUG:Set node state from preCandidate to candidate
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:104) ]
INFO:Send vote request to PeerId{endpoint=localhost:12220, checksum=0} at 400066842 on newTerm 2
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T1][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.election(ElectionServiceImpl.java:104) ]
INFO:Send vote request to PeerId{endpoint=localhost:12230, checksum=0} at 400066842 on newTerm 2
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T3][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handleElectionResponse(ElectionServiceImpl.java:140) ]
INFO:Handle vote response from: raft-3 result: true
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T3][ Class:rpc.RpcServicesImpl Method:
service.ElectionServiceImpl.handleElectionResponse(ElectionServiceImpl.java:148) ]
INFO:Current node start to perform as leader at term 2,grant peer list [raft-1, raft-3]
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T3][ Class:core.NodeImpl Method:
core.NodeImpl.setNodeState(NodeImpl.java:1121) ]
DEBUG:Set node state from candidate to leader
"2020-05-21 13:26:20 上午 [Thread: SOFA-RPC-CB-8-T3][ Class:core.Replicator Method:
core.Replicator.start(Replicator.java:403) ]
INFO:Send emptyAppendEntries request to raft-2 at index 0 on term 2
```

图 23 节点宕机测试日志

Figure 25 Node failure test log

可以看到，新选举出来的节点的 term 由原来的 1 变为 2。此时新一轮日志复制便可开始进行。

6.2 日志复制与状态机应用功能测试结果与分析

本小节将对日志复制以及状态机应用进行相应测试。

如图所示，首先当系统收到客户端请求后，进行封装，并以 LogEvent 的形式发布至 RingBuffer，主节点开始落盘后，将同时发布该事件到 Replicator。

```
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12221-5-T1][ Class:core.NodeImpl Method:
core.NodeImpl.apply(NodeImpl.java:404) ]
INFO:Applying task
"2020-06-02 11:53:57 上午 [Thread: JRaft-NodeImpl-Disruptor--14-thread-1][ Class:core.NodeImpl Method:
core.NodeImpl$LogEntryEventHandler.onEvent(NodeImpl.java:465) ]
DEBUG:Receive logEvent:LogId [index=0, term=0]
"2020-06-02 11:53:57 上午 [Thread: JRaft-NodeImpl-Disruptor--14-thread-1][ Class:core.LogManagerImplNew
Method: core.LogManagerImplNew.appendEntries(LogManagerImplNew.java:97) ]
DEBUG:setLogIndex, first:0 last:0
"2020-06-02 11:53:57 上午 [Thread: JRaft-NodeImpl-Disruptor--14-thread-1][ Class:utils.Recyclers Method:
utils.Recyclers.<clinit>(Recyclers.java:57) ]
DEBUG:-Djraft.recyclers.maxCapacityPerThread: 4096
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.LogManagerImplNew
Method: core.LogManagerImplNew$StableClosureEventHandler.onEvent(LogManagerImplNew.java:291) ]
DEBUG:Send to all replicators with entries sizes: 1 and first index:LogId [index=0, term=2]
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.ReplicatorGroupImpl
Method: core.ReplicatorGroupImpl.lambda$sendAppendEntriesToAllReplicator$0(ReplicatorGroupImpl.java:146) ]
DEBUG:Publish event to ringBuffer localhost:12200 data:1log0 index:LogId [index=0, term=2]
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.ReplicatorGroupImpl
Method: core.ReplicatorGroupImpl.lambda$sendAppendEntriesToAllReplicator$0(ReplicatorGroupImpl.java:146) ]
DEBUG:Publish event to ringBuffer localhost:12230 data:1log0 index:LogId [index=0, term=2]
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:config.LogStorageOptions
Method: core.LogStorageImpl.addDataBatch(LogStorageImpl.java:159) ]
DEBUG:RocksDB put:key:0 value:1log0 length:5
```

图 24 状态机应用测试日志

Figure 26 State machine application test log

Replicator 收到 Log Event 后将进行检查，同时发送给其所负责的 follower。

```
DEBUG:Replicator receive log event on logEntry: LogId [index=0, term=2] data: java.nio.HeapByteBuffer[pos=0
lim=5 cap=5] data:java.nio.HeapByteBuffer[pos=0 lim=5 cap=5]
"2020-06-02 11:53:57 上午 [Thread: JRaft-Replicator-Disruptor--34-thread-1][ Class:core.Replicator Method:
core.Replicator$ReplicatorHandler.onEvent(Replicator.java:196) ]
DEBUG:Replicator send handleAppendEntriesRequests with sizes:1 to localhost:12200
"2020-06-02 11:53:57 上午 [Thread: JRaft-Replicator-Disruptor--37-thread-1][ Class:core.Replicator Method:
core.Replicator$ReplicatorHandler.onEvent(Replicator.java:169) ]
DEBUG:Replicator receive log event on logEntry: LogId [index=0, term=2] data: java.nio.HeapByteBuffer[pos=0
lim=5 cap=5] data:java.nio.HeapByteBuffer[pos=0 lim=5 cap=5]
"2020-06-02 11:53:57 上午 [Thread: JRaft-Replicator-Disruptor--37-thread-1][ Class:core.Replicator Method:
core.Replicator$ReplicatorHandler.onEvent(Replicator.java:196) ]
DEBUG:Replicator send handleAppendEntriesRequests with sizes:1 to localhost:12230
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.NodeImpl Method:
core.NodeImpl$LeaderStableClosure.run(NodeImpl.java:776) ]
DEBUG:LeaderStableClosure:Status{firstIndex=0, lastIndex=0, state=null} first:0 last:0
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.NodeImpl Method:
entity.BallotBox.<init>(BallotBox.java:50) ]
DEBUG:BallotBox init success with quorum 2    at index 0 and peerList [PeerId{endpoint=localhost:12200,
checksum=0}, PeerId{endpoint=localhost:12230, checksum=0}, PeerId{endpoint=localhost:12220, checksum=0}]
"2020-06-02 11:53:57 上午 [Thread: JRaft-LogManager-Disruptor--20-thread-1][ Class:core.NodeImpl Method:
entity.BallotBox.grant(BallotBox.java:60) ]
DEBUG:log at 0 length:1 was granted by raft-2, current ballot status:false    current grant list
[PeerId{endpoint=localhost:12220, checksum=0}]
```

图 25 状态机应用测试日志

Figure 27 State machine application test log

Leader 在收到 LeaderStableClosure 后开始新建投票箱 BallotBox，该投票箱将负责该日志 index 的投票工作，以确保在收到 Follower 同意的情况下，能够决定该日志是否 Commit。

```
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T7][ Class:core.NodeImpl Method:
entity.BallotBox.grant(BallotBox.java:60) ]
DEBUG:log at 0 length:1 was granted by raft-3, current ballot status:true    current grant list
[PeerId{endpoint=localhost:12220, checksum=0}, PeerId{endpoint=localhost:12200, checksum=0}]
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T8][ Class:core.NodeImpl Method:
entity.BallotBox.grant(BallotBox.java:60) ]
DEBUG:log at 0 length:1 was granted by raft-1, current ballot status:true    current grant list
[PeerId{endpoint=localhost:12220, checksum=0}, PeerId{endpoint=localhost:12200, checksum=0}]
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T7][ Class:core.NodeImpl Method:
entity.BallotBox.apply(BallotBox.java:132) ]
INFO:Ballot box invokes apply at index 0 length 1 with grant list [PeerId{endpoint=localhost:12220, checksum=0},
PeerId{endpoint=localhost:12200, checksum=0}]
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T8][ Class:core.NodeImpl Method:
entity.BallotBox.apply(BallotBox.java:132) ]
INFO:Ballot box invokes apply at index 0 length 1 with grant list [PeerId{endpoint=localhost:12220, checksum=0},
PeerId{endpoint=localhost:12200, checksum=0}]
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T7][ Class:core.FSMCallerImpl Method:
core.FSMCallerImpl.onCommitted(FSMCallerImpl.java:181) ]
DEBUG:onCommitted the log at committedIndex 0
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T8][ Class:core.FSMCallerImpl Method:
core.FSMCallerImpl.onCommitted(FSMCallerImpl.java:181) ]
DEBUG:onCommitted the log at committedIndex 0
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12220-3-T7][ Class:core.ReplicatorGroupImpl Method:
core.ReplicatorGroupImpl.sendApplyNotifyToAll(ReplicatorGroupImpl.java:113) ]
DEBUG:Send to all replicator that the log index:0 can be applied
```

图 26 状态机应用测试日志

Figure 28 State machine application test log

当收到来自大多数节点的日志同意请求后，BallotBox 将能够确定该日志可以应用，并交由 FSMCaller 执行状态机应用，同时通知其他节点开始应用该日志到状态机。

```
2020-06-02 11:53:57 上午 [Thread: JRaft-FSMCaller-Disruptor--23-thread-1][ Class:core.FSMCallerImpl Method:
core.FSMCallerImpl.doCommitted(FSMCallerImpl.java:153) ]
INFO:doCommitted at 0 iterImpl:IteratorImpl [fsm=core.CustomStateMachine@5c3b9846,
logManager=core.LogManagerImplNew@58040719, currentIndex=0, committedIndex=0,
currEntry=entity.LogEntry@52fc6c69, applyingIndex=0, error=null,isGood=true] isGood:true
"current:0 committedIndex:0
2020-06-02 11:53:57 上午 [Thread: JRaft-FSMCaller-Disruptor--23-thread-1][ Class:core.FSMCallerImpl Method:
core.FSMCallerImpl.doApplyTasks(FSMCallerImpl.java:169) ]
INFO:Apply to fsm at 0
"2020-06-02 11:53:57 上午 [Thread: JRaft-FSMCaller-Disruptor--23-thread-1][ Class:core.CustomStateMachine
Method: core.CustomStateMachine.onApply(CustomStateMachine.java:27) ]
DEBUG:onApply:0 operation:false
"current:0 committedIndex:0
2020-06-02 11:53:57 上午 [Thread: JRaft-FSMCaller-Disruptor--23-thread-1][ Class:core.CustomStateMachine
Method: core.CustomStateMachine.onApply(CustomStateMachine.java:33) ]
DEBUG:CustomStateMachine handle LOG:0
"2020-06-02 11:53:57 上午 [Thread: JRaft-FSMCaller-Disruptor--23-thread-1][ Class:core.NodeImpl Method:
core.NodeImpl.handleLogApplied(NodeImpl.java:843) ]
INFO:Log 0 has been applied to stateMachine
```

图 29 状态机应用测试日志

Figure 29 State machine application test log

执行到状态机的过程即是封装成 Iterator 交由 StateMachine 进行 apply，同时设置 StableLog 为当前应用到的最高 Index。

6.3 线性一致读测试结果与分析

本小节将针对线性一致读进行测试。首先系统接受到来自客户端的请求，该请求将被封装为 ReadTask 且以 Event 的形式发布于 Ringbuffer。对应的 Event 处理方将当前的 StableLogIndex 与 ReadTask 封装为 ReadTaskSchedule，等待心跳包返回确定是 Leader 的时候取出。取出后交由状态机进行读取操作，此后异步通知客户端线性一致读结果。

```
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12221-5-T1][ Class:core.NodeImpl Method:
core.NodeImpl.apply(NodeImpl.java:404) ]
INFO:Applying Read task
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12221-5-T1][ Class:core.NodeImpl Method:
core.NodeImpl.apply(NodeImpl.java:404) ]
INFO:Add readEvent
```

图 27 线性一致读测试日志

Figure 30 Linearable reading test log

```
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12299-3-T1][ Class:rpc.ClientRpcServiceImpl Method:
rpc.ClientRpcServiceImpl.readResult(ClientRpcServiceImpl.java:33) ]
INFO:Receive ReadTask entity.ReadTask@42fe0aa6
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12299-3-T1][ Class:rpc.ClientRpcServiceImpl Method:
rpc.ClientRpcServiceImpl.readResult(ClientRpcServiceImpl.java:35) ]
INFO:Read result:1log0
"2020-06-02 11:53:57 上午 [Thread: SOFA-SEV-BOLT-BIZ-12299-3-T2][ Class:rpc.ClientRpcServiceImpl Method:
rpc.ClientRpcServiceImpl.readResult(ClientRpcServiceImpl.java:33) ]
INFO:Receive ReadTask entity.ReadTask@58dc55d0
```

图 28 线性一致读测试日志

Figure 31 Linearable reading test log

可以看到，线性一致读的结果为正确写入的结果，本论文所提出的系统在经过正常选举，领导者宕机后选举等情况后仍能正常读取到结果。此测试反映出 Raft 协议优良的容错性。

6.4 压力测试结果与分析

本小节将进行压力测试，测试该系统最大的 TPS 值。TPS（Transactions Per Second）每秒事务数，在本文中，一次写入操作被视为一个事务。客户端将运行 1000ms，并将所有成功请求数据记为 TPS[29]。测试条件与测试结果如下

表格 4 压力测试结果

Table 4 Benchmark results

Client 数量	Client -Batching	Storage -Type	Replicator-Pipeling	Key 大小	Value 大小	通信方式	序列化协议	TPS
1	否	RocksDB	是	4B	4B	RPC 异步	内部：Protobuf 业务：Hessian	685
1	否	RocksDB	是	4B	4B	RPC 异步	内部：Protobuf 业务：Hessian	890
4	两个是，两个否	RocksDB	是	4B	4B	RPC 异步	内部：Protobuf 业务：Hessian	1123

通过压力测试发现，在开启 Client-Batching 的系统上性能效果要优于未开启的系统，且系统优化空间较大，请求方式的改变对最终系统性能具有一定的影响。后续需要对其他细节进行持续优化以保证性能不断提升[30]。此外根据日志分析得知，Raft 协议要求日志应用必须按日志复制的顺序进行，部分日志复制的延迟会一定程度上拖慢后续日志的应用。

6.5 本章小结

本章先从功能测试开始，从选举，日志复制，状态机应用与线性一致读方面进行基础功能测试，同时还完成了节点宕机情况下重新选举的容错性测试。经过测试可以验证该系统具有 Raft 协议所要求的基本功能，且能够在保证系统稳定运行的情况下承受一定程度的节点宕机。随后压力测试从多个角度，分别测试了使用 Client-Batching 的不同情况下所能达到的 TPS，经过测试可见，使用 Client-Batching 可以有效提升系统的负载，其主要原理是通过减少网络请求，通过一次请求多个 log 的方式达成目标。

第七章 总结与展望

7.1 工作总结

本论文设计并实现了一种基于 Raft 协议具有容错性且高效运行的分布式一致性存储系统。该系统架构和部分流程参考阿里巴巴的 SOFAJRaft。针对现有实现系统中存在的日志复制、存储、打包效率低下，以及通信连接不稳定造成系统高故障率等问题，提出并实现了相应的优化策略。最后，使用不同优化策略、不同数量的客户端对系统进行了基本功能测试与压力测试。本文所取的的成果与不足总结如下：

除网络通信与持久化操作外，所有 Raft 协议相关内容独立实现并优化。从领导人选举到日志复制，再到状态机应用与线性一致读，均由本文作者自 2020 年 3 月 18 日自 2020 年 5 月 21 日开发完成。本文所有代码将在论文终稿完成后开源于 GitHub 平台，地址见本文文末附件。在系统实现过程中针对日志复制与状态机应用过程中可能出现的问题，本文参考业界优秀实现并结合本系统实际情况进行优化。主要包括日志复制 Pipeline 与批量应用。此举有效减少了等待时间，加快了系统的运行效率。同时本文测试章节均在阿里云 ECS 抢占式实例中进行，模拟三节点分布式系统。测试结果对今后的改进与优化具有极大帮助。

本文不足之处主要包括理解不透彻，后续功能缺失以及容错性不够高等。

因为时间限制，本次仅阅读了 Raft 作者针对 Raft 的论文的精简版，并未阅读 Raft 协议的全文。因此少数功能理解还不够透彻，部分实现细节可能与论文所述相违背，需要后续继续学习与理解。且本系统未能实现 Raft 论文中提到的快照（SnapShot）与配置变更（Membership Changes），需要在后期学习过程中进行实现。本文所提出系统在设计时便要求具有一定的容错性，以便达到分布式系统要求。在测试过程中也确实涵盖了部分容错性测试。但是针对容错性方面的测试还远不及分布式环境复杂，需要根据多种情况与场景进行一步测试。并针对测试结果进行容错性支持。

7.2 展望

通过对 Raft 协议的实现，可以了解到一个设计优秀的分布式系统不仅要在共识协议上保证高效可靠，更需要在设计细节上下足功夫。经过测试，对今后的工作重点与展望如下：

完善快照存储（Snapshot）的设计与实现。快照对在实现了 Raft 协议的分布式系统提升效率至关重要，能够缩短节点跟上主节点的速度，且有利于减轻网络压力。

完善配置变更（Configuration Changes）的设计与实现。因配置变更是运行 Raft 协议的分布式系统的长久稳定运行的必要保障，可以保证在不关停系统的情况下保证对成员信息的变更与更改。

完善更广范围的容错性测试，包括日志复制与状态机应用的容错测试，确保节点可以在任意时间段发生故障而不影响整体系统的正常运行。同时要根据压测结果从多个细节方面进行优化，确保系统性能发挥的更极值。

参考文献

[1]曾诗钦. 北京邮电大学网络与交换国家重点实验室, 区块链技术研究综述：原理、进展与应用[J],2020.

[2] 王海勇; 郭凯璇; 潘启青. 南京邮电大学计算机学院, 基于投票机制的拜占庭容错共识算法[J], 2019.

[3] 阳振坤. 蚂蚁金融服务集团, 分布式关系数据库 OceanBase 的高可靠性[J], 2016.

[4]苗凡；阎志远. 中国铁道科学研究院集团有限公司，基于 Zookeeper 的配置管理中心设计与实现[J], 2018.

[5]张仕将;柴晶. 太原理工大学，中国科学与计算机网络信息中心，基于 Gossip 协议的拜占庭共识算法[J],2018.

[6]郭奕庭. 东南大学, 基于 Node.js 与 Etcd 的轻量级微服务开发平台的设计与实现[D], 2018.

[7]Alibaba Ant Finanical Services Group. SOFAJRaft GitHub[CP]. https://github.com/sofastack/sofa-jraft 2019.

[8]LESLIE LAMPORT, Microsoft, Paxos Made Simple[J], 2001.

[9]Diego Ongaro and John Ousterhout. Stanford University. In Search of an Understandable Consensus Algorithm[D], 2013.

[10] 鲁子元. 黑龙江省网络空间研究中心, 浅析 RAFT 分布式算法[J], 2017.

[11] 王斌斌. 电子科技大学, 基于 Protobuf 的 RPC 系统的设计与实现[D], 2016.

[12] 邹文静. 电子科技大学, 基于 RocksDB 引擎的分布式存储系统设计与实现[D], 2018.

[13] LAMX. LAMX Disruptor, https://lmax-exchange.github.io/disruptor/[EB/OL], 2020.

[14] MeiTuan. 高性能队列 Disruptor 美团技术团队 [EB/OL], https://tech.meituan.com/2016/11/18/disruptor.html, 2018.

[15] 崔政; 段利国. 太原理工大学, 基于 Java synchronized 同步锁实现线程交互[J],2018.

[16] 盛琳阳; 盛芳圆. 黑龙江工商学院, Java 多线程技术在网络通信系统中的应用[J],2019.

[17] 李丹; 叶廷东. “异地多活”分布式存储系统设计和实现[J], 2020.

[18] Alibaba Ant Finanical Services Group. SOFAJRaft 选举机制剖析 [EB/OL], https://www.sofastack.tech/blog/sofa-jraft-election-mechanism/, 2019.

[19] Alibaba Ant Finanical Services Group. SOFAJRaft 日志复制 - pipeline 实现剖析 [EB/OL] https://www.sofastack.tech/blog/sofa-jraft-pipeline-principle/, 2019.

[20] M Mavronicolasy. University of Cyprus, Linearizable Read/Write Objects[J],1998.

[21] Alibaba Ant Finanical Services Group. SOFAJRaft 线性一致读实现剖析 [EB/OL] https://www.sofastack.tech/blog/sofa-jraft-linear-consistent-read-implementation/, 2019.

[22] 李迪. 北京邮电大学，基于 Raft 协议及 RocksDB 的分布式统一配置中心设计与实现[D], 2019.

[23] 张晨东;郭进伟. 华东师范大学，基于 Raft 一致性协议的高可用性实现[J], 2018.

[24] Alibaba Ant Finanical Services Group. Logs in memory needs a more appropriate deque. https://github.com/sofastack/sofa-jraft/issues/335[Z], 2019.

[25] 陈陆. 江苏科技大学, 分布式键值存储引擎的研究与实现[D], 2018.

[26] Alibaba Ant Finanical Services Group. Raft Introduction[EB/OL], https://www.sofastack.tech/projects/sofa-jraft/raft-introduction/, 2019

[27] 钱迎进; 肖侬; 金士尧. 国防科技大学，大规模集群中一种自适应可扩展的 RPC 超时机制[J], 2011

[28] 万勇, 华中科技大学, 集群系统中的网络性能优化方法研究[D], 2013.

[29] 吴维; 唐永强; 任喜亮. 上海游族信息技术有限公司, 服务器性能压力测试的加压数据复用方法[P], 2020.

[30] Alibaba Ant Finanical Services Group. Benchmark 数据[J], https://www.sofastack.tech/projects/sofa-jraft/benchmark-performance/, 2019.

致谢

最初选择这个论文题目的时候，我其实十分担心我能否顺利完成。在这里十分感谢我的导师冯老师，在选题时给予我最大的支持与自由，并在后来的论文编写与毕业设计制作过程中提供重要建议。

一路走来，得益于信息学院科技创新协会提供的平台和资源，让我能够接触到优秀的指导教师与水平高超的各位学长。在此感谢为信息学院科协提供多年指导的周老师。同时也十分感谢仲学长在我入驻科协后给予我的引导与严格要求，以及王学长、dandan 学长、刘 cc 学长等科协前辈们在我刚进入科协时给予我的理解与支持。

在我大学的第三年十分荣幸的担任了信息学院科协的会长。一年的任期内，各位副会长也给予了我工作上的极大支持，借此机会感谢各位我身边的副会长大佬们：lmy 同学，cjn 同学，snf 同学，lzk 同学，fxk 同学等。与此同时也要感谢时任的各位组长以及组员，工作的顺利开展离不开各位的共同努力。具体科协实验室成员名单将在附件中体现。最后还要感谢当年支持科协工作的各位学生会主席，以及青科大学生会权益中心 QingSo 团队。

大学的第四年，我终于遇到了我的小可爱——zxw 同学，我十分生气为什么她不早点来，以至于我在大学最后一年才体会到有女朋友的幸福与开心。尽管 xw 才大一，我已经大四了，但是我们还是磨合的很好。我生病的时候 xw 到宿舍楼下帮我贴冰宝宝，我讲课的时候 xw 帮我带饭在门外等我下课…… 写这篇论文的时候，花了许多时间，没能好好陪她，现在想起来真的觉得心里很对不起自己的女朋友。写到这里，我已经和 xw 有 140 天没有见面了，交完论文完成毕业手续我一定要去见她，好好解释一下论文没有女朋友重要，140 天没见面其实全怪有人吃蝙蝠。

关于未来，我其实还有人要感谢，我打算到地球的另一边去学习。十分感谢当初支持我做决定的爸爸妈妈，还有帮我填筹备申请的张老师和李老师，以及当初帮我筹备雅思考试的潘老师，Tiffany，Angela，还有王老师。最后还要感谢 wlx 同学和我一起背了半年的单词，最后考试的时候有的单词还用上了。

最后，絮絮叨叨的我还有好多要感谢的人，我的舍友：zjq、zqq、ldy、gxg、chk 在我每天晚上忙的时候回来很晚也给我很大理解，一些需要撑场的场合几个兄弟一个没少过。我的朋友，xjc，在平时生活中给了我不少建议，他的建议让我看待事物更全面更具体，做事情也更有方向。还要感谢 lzk，pc 学长，dxy 学姐让我学到了很多现实社会上的事情。另外也要顺带感谢一下信息学院党总支副书记王军对我最后半年工作的严格要求以及最后对我工作不优秀的评价，祝王军副书记今后工作顺利。

要感谢的人太多，恐怕写很久也写不完，未上致谢的人也不是没有感激，而是来日方长。也再次感谢所有帮助过我却未出现在致谢中人，我也十分原意在今后的日子里为大家排忧解难，一同前行。

至此，我的大学四年即将画上句号，这一路经历太多，也让我变得愈发坚强。回头看看来时走的路，再多的回忆，感谢，也抵不过再走一次的幸福。

2020 年 5 月 30 日 0:39

于山东威海

附录

论文所提出系统开源代码地址

https://github.com/CoderiGenius/RocksRaft