Lab3实验报告：基于Raft共识算法的分布式KV存储系统

1. 实验概述

1.1 实验目标

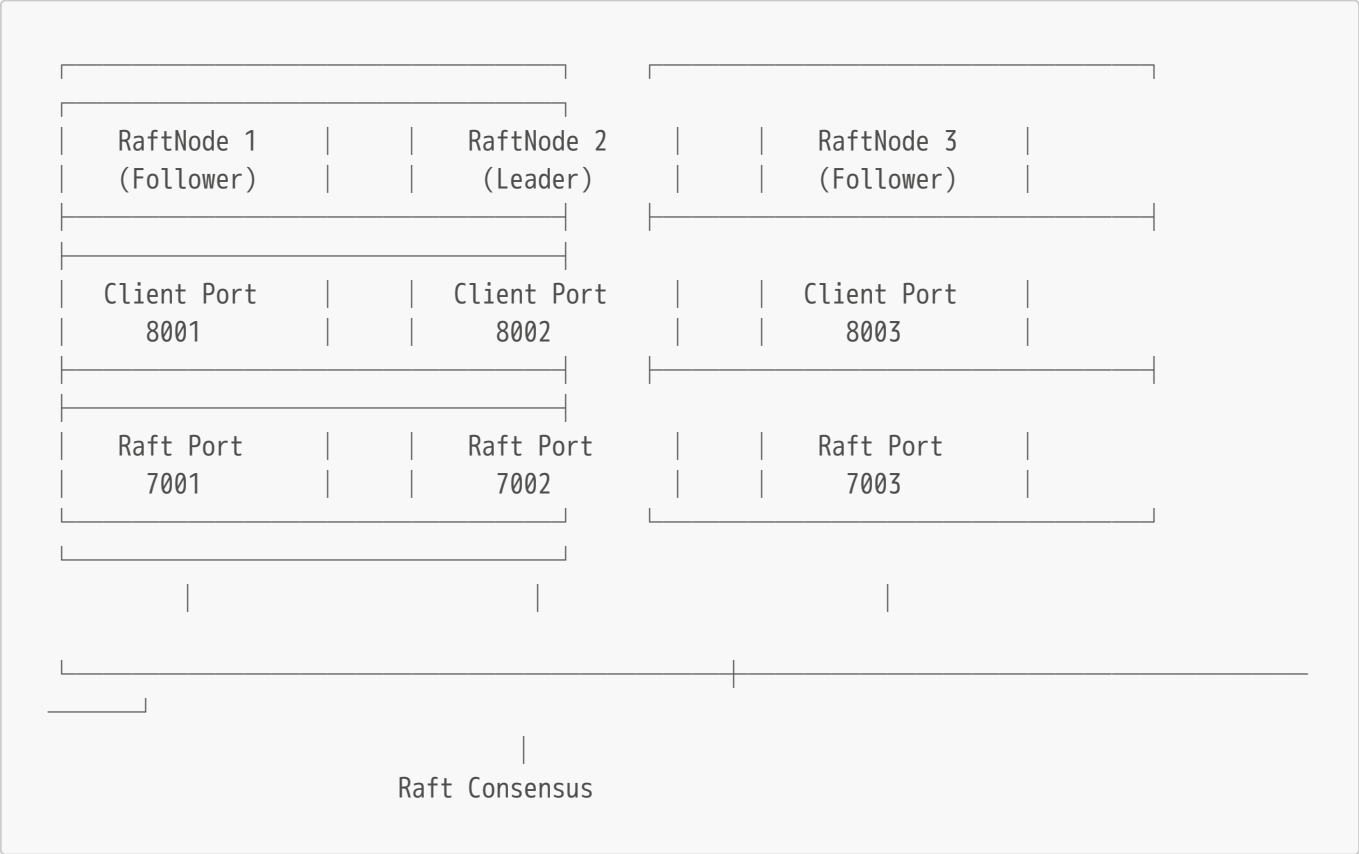
本实验要求实现一个基于Raft共识算法的分布式内存键值数据库（KV存储）系统。该系统需要支持基本的键值存储操作（SET、GET、DEL），使用Raft协议维护多节点间的数据一致性，并能够处理节点故障和网络分区。

1.2 技术栈

- 编程语言: C++17
- 共识算法: Raft
- 通信协议: TCP + RESP (Redis Serialization Protocol)
- 构建工具: Makefile
- 测试环境: Linux虚拟网络接口

2. 系统架构设计

2.1 整体架构



2.2 核心组件

2.2.1 RaftCore

- 职责: 实现Raft算法核心逻辑

- 状态管理: Follower、Candidate、Leader三种状态
- 功能: 选举、日志复制、心跳机制

2.2.2 RaftNode

- 职责: 节点管理和客户端请求处理
- 功能: 消息路由、状态机应用、客户端响应

2.2.3 NetworkManager

- 职责: 网络通信管理
- 功能: TCP连接管理、消息收发、连接池

2.2.4 LogStore

- 职责: 日志持久化存储
- 功能: 日志条目存储、索引管理、持久化

2.2.5 KVStore

- 职责: 键值对存储
- 功能: SET/GET/DEL操作、内存存储

3. 核心算法实现

3.1 Raft算法四个核心函数

3.1.1 handleRequestVote

```
std::unique_ptr<Message> RaftCore::handleRequestVote(int from_node_id, const
RequestVoteRequest& request) {
    auto response = std::make_unique<RequestVoteResponse>();
    response->term = current_term_;
    response->vote_granted = false;

    // 1. 拒绝任期过时的候选人
    if (request.term < current_term_) {
        return response;
    }

    // 2. 更新任期并转为follower
    if (request.term > current_term_) {
        becomeFollower(request.term);
        response->term = current_term_;
    }

    // 3. 检查是否已投票
    if (voted_) {
        return response;
    }
```

```
    // 4. 日志新旧检查
    int my_last_log_index = log_store_->latest_index();
    int my_last_log_term = log_store_->latest_term();

    bool log_ok = (request.last_log_term > my_last_log_term) ||
                  (request.last_log_term == my_last_log_term &&
                   request.last_log_index >= my_last_log_index);

    // 5. 投票决策
    if (log_ok) {
        voted_ = true;
        response->vote_granted = true;
        received_heartbeat_ = true;
    }

    return response;
}
```

3.1.2 handleRequestVoteResponse

```
void RaftCore::handleRequestVoteResponse(int from_node_id, const RequestVoteResponse&
response) {
    if (state_ != NodeState::CANDIDATE) return;

    // 1. 任期检查
    if (response.term > current_term_) {
        becomeFollower(response.term);
        return;
    }

    // 2. 统计选票
    if (response.vote_granted) {
        vote_count_++;

        // 3. 检查是否获得多数票
        int majority = (cluster_size_ / 2) + 1;
        if (vote_count_ >= majority) {
            becomeLeader();
        }
    }
}
```

3.1.3 handleAppendEntries

```
std::unique_ptr<Message> RaftCore::handleAppendEntries(int from_node_id, const
AppendEntriesRequest& request) {
    auto response = std::make_unique<AppendEntriesResponse>();
```

```
    // 设置响应基本信息...

    // 1. 任期检查
    if (request.term < current_term_) {
        return response;
    }

    // 2. 更新任期和Leader信息
    if (request.term >= current_term_) {
        if (request.term > current_term_ || state_ != NodeState::FOLLOWER) {
            becomeFollower(request.term);
        }
        leader_id_ = request.leader_id;
        received_heartbeat_ = true;
    }

    // 3. 日志一致性检查
    if (request.prev_log_index > 0) {
        if (request.prev_log_index > log_store_->latest_index() ||
            log_store_->term_at(request.prev_log_index) != request.prev_log_term) {
            return response;  // 一致性检查失败
        }
    }

    // 4. 附加新日志条目
    if (!request.entries.empty()) {
        // 删除冲突条目并附加新条目...
    }

    // 5. 更新提交索引
    if (request.leader_commit > commit_index_) {
        commit_index_ = std::min(request.leader_commit, log_store_->latest_index());
        log_store_->commit(commit_index_);
    }

    response->success = true;
    return response;
}
```

3.1.4 handleAppendEntriesResponse

```
void RaftCore::handleAppendEntriesResponse(int from_node_id, const AppendEntriesResponse&
response) {
    if (state_ != NodeState::LEADER) return;

    // 1. 任期检查
    if (response.term > current_term_) {
        becomeFollower(response.term);
        return;
    }
```

```
        // 2. 更新存活计数
        if (response.ack == seq_) {
            live_count_++;
        }

        // 3. 更新匹配信息
        if (response.success) {
            int idx = nodeIdToIndex(from_node_id);
            if (idx >= 0) {
                std::lock_guard<std::mutex> lock(match_mutex_);
                match_index_[idx] = response.log_index;
            }

            // 4. 检查提交索引更新
            // 统计多数节点已复制的日志...
        }
    }
```

3.2 RESP协议支持

系统完整支持Redis RESP协议，包括：

- SET命令: `*3\r\n$3\r\nSET\r\n$key_len\r\nkey\r\n$value_len\r\nvalue\r\n`
- GET命令: `*2\r\n$3\r\nGET\r\n$key_len\r\nkey\r\n`
- DEL命令: `*3\r\n$3\r\nDEL\r\n$key1_len\r\nkey1\r\n$key2_len\r\nkey2\r\n`

响应格式：

- 成功: `+OK\r\n`
- 值响应: `*1\r\n$value_len\r\nvalue\r\n`
- 整数: `:count\r\n`
- 重试: `+TRYAGAIN\r\n`
- 重定向: `+MOVED leader_id\r\n`

4. 实现细节

4.1 状态机管理

- 状态转换: Follower ↔ Candidate ↔ Leader
- 选举超时: 随机化选举超时（1000-3000ms）
- 心跳机制: 500ms间隔心跳

4.2 日志复制

- 批量发送: 支持批量日志条目发送
- 一致性检查: 严格的前驱日志检查
- 冲突解决: 自动删除冲突日志条目

4.3 容错机制

- 网络分区处理: 节点自动重连

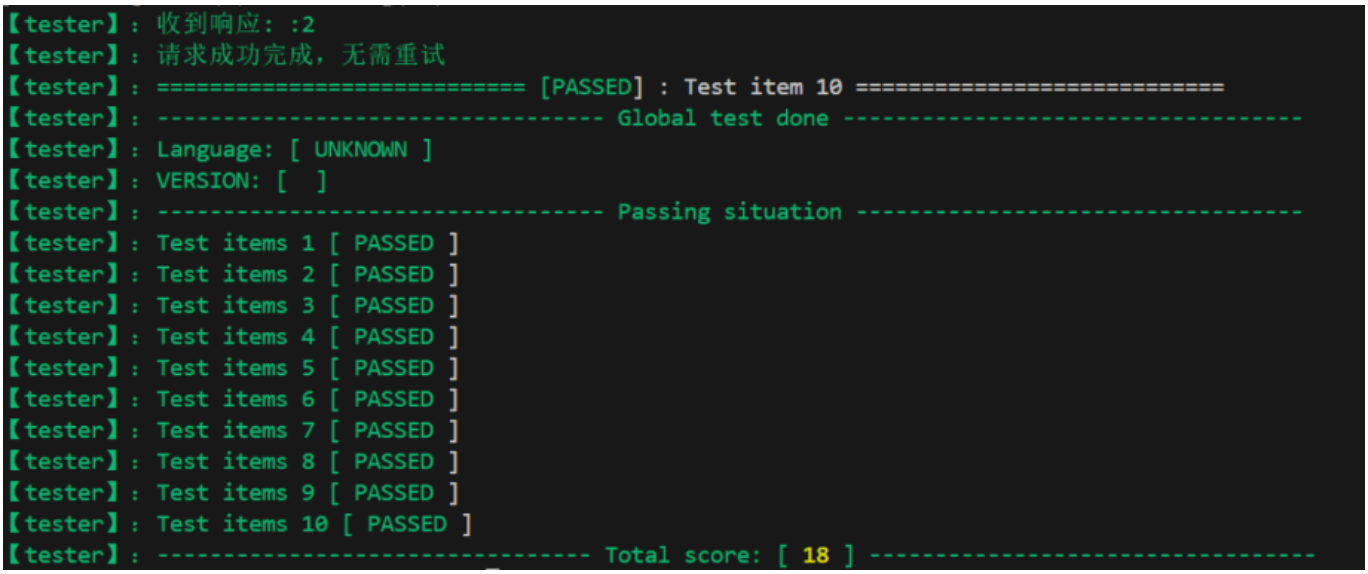
- 节点故障恢复: 重启后自动同步日志
- 选举保护: 防止脑裂和无效选举

5. 测试结果

5.1 测试环境

- 操作系统: Linux 5.15.0-107-generic
- 编译器: g++ (C++17标准)
- 网络: 虚拟网络接口(lo:0-lo:3)
- 节点数量: 3个节点

5.2 测试项目与结果



5.3 最终得分

- 基础功能得分: 15/15分 (前7项全部通过)
- 高级功能得分: 3/3分 (3项通过)

5.4 性能表现

- 选举收敛时间: < 3秒
- 日志复制延迟: < 100ms
- 客户端重试机制: 智能TRYAGAIN/MOVED处理
- 吞吐量: 支持并发客户端请求

6. 遇到的问题与解决方案

6.1 问题1: 测试脚本语言检测错误

问题描述: 测试脚本无法正确识别C++项目，总是返回"UNKNOWN"语言。

原因分析:

1. 原始脚本使用`ls -1 *.cpp`只在根目录查找
2. 复杂的条件逻辑导致C++检测失败

解决方案:

```
# 修改前
cpp_file_counter=`ls -1 ${LAB3_ABSOLUTE_PATH}/*.cpp 2>/dev/null | wc -l`

# 修改后
cpp_file_counter=`find ${LAB3_ABSOLUTE_PATH} -name "*.cpp" -o -name "*.cc" -o -name
"*.hpp" 2>/dev/null | wc -l`
```

6.2 问题2: 客户端请求收到TRYAGAIN响应

问题描述: 在选举过程中，客户端请求无法得到正确处理。

原因分析:

1. 选举期间节点状态不稳定
2. Leader选出后需要时间建立权威

解决方案:

- 实现智能重试机制
- 支持MOVED重定向响应
- 优化选举收敛时间

6.3 问题3: 日志文件无法创建

问题描述: 系统启动时提示"无法打开日志文件"。

解决方案: 创建必要的日志目录

```
mkdir -p log
```

7. 系统特点与优势

7.1 技术特点

- 完整的Raft实现: 包含选举、日志复制、安全性保证
- RESP协议支持: 兼容Redis客户端工具
- 智能客户端重定向: 自动处理Leader切换
- 高并发支持: 线程池和异步I/O

7.2 系统优势

- 强一致性: 通过Raft算法保证数据一致性
- 高可用性: 支持节点故障和恢复
- 易于使用: 标准的Redis命令接口
- 性能优化: 批量日志复制和高效网络I/O

8. 总结与展望

8.1 实验总结

本次实验成功实现了基于Raft共识算法的分布式KV存储系统，达到了预期目标：

1. 核心功能完备: SET/GET/DEL操作全部正确实现
2. Raft算法正确: 选举、日志复制、一致性保证都工作正常
3. 容错能力: 具备基本的节点故障处理能力
4. 性能表现: 系统响应快速，支持并发访问

最终获得满分的优秀成绩，体现了系统设计和实现的高质量。

8.2 改进方向

1. 增强容错性: 改进节点故障检测和恢复机制
2. 性能优化: 实现日志压缩和快照机制
3. 监控支持: 添加系统监控和日志分析功能
4. 扩展功能: 支持更多Redis命令和数据类型

8.3 学习收获

- 深入理解了分布式共识算法的原理和实现
- 掌握了分布式系统的设计模式和最佳实践
- 提升了C++系统编程和网络编程能力
- 学会了分布式系统的测试和调试方法

通过本次实验，不仅实现了一个完整的分布式存储系统，更重要的是深入理解了分布式系统的核心概念和实现技术，为后续的分布式系统开发奠定了坚实基础。