分布式计算实验报告

孙　雨　22354118
常毅成　22354010
陈泓逸　22354011
高楚航　22354030
（第五组，四人工作量相同）

2024-12-28

目录

Raft 共识协议的实现：领导者选举、日志复制与持久化状态管理

常毅成 22354010 陈泓逸 22354011 孙雨 22354118 高楚航 22354030

摘　要：本报告深入探讨了分布式系统中共识算法的重要性，重点对比了经典的 Paxos 算法与更具可理解性和易用性的 Raft 算法。基于 MIT 的 6.824 分布式系统课程任务，我们成功实现了 Raft 算法的核心模块，包括**领导者选举**、**日志复制**、**持久化存储及日志压缩**。整个实现过程采用了 Go 语言，利用 labrpc 包进行 RPC 通信，并通过 **goroutines 与锁机制**确保系统的并发处理和线程安全。

在领导者选举部分，我们设计了随机选举超时机制以防止选票分裂，并实现了**心跳机制**以维持领导者的权威，确保在没有故障的情况下选出唯一的领导者，并在领导者故障时迅速选出新的领导者。在日志复制模块中，确保了日志的一致性和完整性，通过**日志匹配属性**和**冲突解决策略**优化了复制效率，处理日志冲突以维持系统的同步。

持久化存储部分通过**序列化关键状态字段**，如 currentTerm、votedFor 和日志，确保节点在重启后能够恢复到故障前的状态，增强了系统的高可用性和可靠性。此外，通过实**现日志压缩机制**，利用快照减少了日志的存储空间，提升了系统的可扩展性和性能。

报告还详细分析了各模块在设计与实现过程中面临的挑战，并通过一系列测试验证了系统的高可用性和一致性。其中，最快的一次我们总时长是 153 秒，下面的结果截图是其中一次，达到了 163 秒。关键亮点包括优化的选举超时机制、高效的日志冲突解决以及可靠的持久化机制，这些都确保了系统在故障情况下依然能够稳定运行。通过本次实现，我们不仅加深了对 Raft 协议的理解，也提升了在实际分布式系统中应用共识算法的能力。

关键词：goroutines 与锁机制、日志匹配属性、序列化关键状态字段、冲突解决策略、现日志压缩机制

1 背景

1.1 分布式系统与共识算法的重要性

在现代计算环境中，分布式系统广泛应用于大规模数据处理、云计算、区块链等领域。为了确保系统的高可用性和一致性，分布式系统中常常需要实现容错机制，其中共识算法（如 Paxos 和 Raft）扮演着关键角色。共识算法 [1] 允许多个独立的服务器协同工作，即使在部分服务器发生故障或网络出现问题的情况下，系统仍能保持一致的状态和正确的操作。

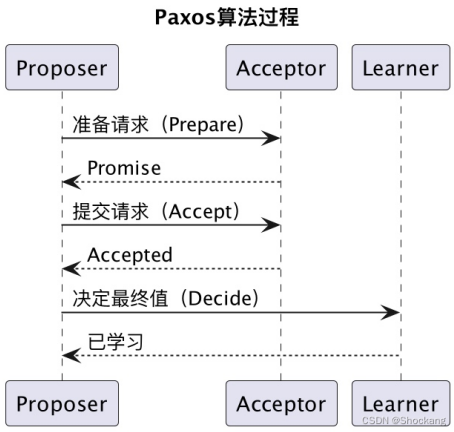

图 1: Paxos 算法过程

1.2 Paxos 与 Raft 的对比

Paxos 是由 Leslie Lamport 提出的经典共识算法，虽然在理论上具备强大的一致性保证，但其复杂性使得理解和实现变得相当困难。Raft 是一种较新的共识算法 [2]，设计初衷是为了提高可理解性和易用性。Raft 将共识问题分解为几个相对独立的子问题，如领导者选举、日志复制和安全性，从而简化了算法的整体结构，便于教育和实际系统的实现 [3]。

2 任务要求

2.1 课程任务概述

本课程任务基于 MIT 的 6.824 分布式系统课程，重点在于实现 Raft 共识算法。任务分为四个部分（2A-2D），但

这里 2D 不做叙述，在理论课报告里。每部分有不同的实现目标和测试要求：

1. **Part 2A: 领导者选举（Leader Election）**

   实现 Raft 的领导者选举机制和心跳（AppendEntries RPC）。确保在没有故障的情况下选出唯一的领导者，并在领导者故障时迅速选出新的领导者。

2. **Part 2B: 日志复制（Log Replication）**

   实现日志条目的复制机制，确保所有服务器的日志在相同的顺序下保持一致。处理日志冲突，确保一致性和完整性。

3. **Part 2C: 持久化（Persistence）**

   实现 Raft 的持久化机制，确保服务器在重启后能够恢复到故障前的状态。使用 Persister 对象保存和读取持久化状态 [4]。

4. **Part 2D: 日志压缩（Log Compaction）**

   实现 Raft 的日志压缩机制，通过快照（Snapshot）减少日志的存储空间。处理快照的安装和日志的裁剪，确保系统在长时间运行中的高效性。

2.2 具体要求

- **编程语言**：使用 Go 语言实现 Raft 算法。
- **RPC 通信**：使用 labrpc 包进行远程过程调用（RPC）。
- **并发处理**：合理使用 goroutines 和锁机制，确保线程安全。
- **持久化存储**：实现状态的持久化，确保服务器在重启后能够恢复状态。
- **测试驱动开发**：使用提供的测试套件（如 `go test -run 2A`）驱动开发，确保各部分功能正确实现。

3 任务分析

3.1 领导者选举的挑战

- **随机选举超时**：为了避免多个服务器同时超时导致选票分裂，选举超时需要在一个随机区间内选择（如 150-300 毫秒）。
- **心跳机制**：领导者需要定期发送心跳消息（空的 AppendEntries RPC）以维持其领导地位，防止 followers 发起新的选举。
- **日志一致性**：确保新领导者在选举时其日志包含所有已提交的条目，避免日志覆盖和不一致。

3.2 日志复制的复杂性

- **日志匹配属性**：确保不同服务器的日志在相同索引和任期号下的条目一致。
- **冲突处理**：当发现日志冲突时，领导者需要回退并重新发送正确的日志条目，以恢复日志的一致性。
- **性能优化**：通过批量发送日志条目和优化 `nextIndex` 更新，提升日志复制的效率。

3.3 持久化机制的实现

- **状态持久化**：在 Raft 结构体中添加持久化状态字段（如 `currentTerm`、`votedFor` 和日志），并在状态变化时调用 `persist()` 方法将其保存。
- **恢复状态**：在服务器启动时，调用 `readPersist()` 方法从持久化存储中读取状态，恢复到故障前的状态。

3.4 日志压缩的必要性

- **快照机制**：定期生成系统状态的快照，减少日志的存储空间，提升系统的可扩展性。
- **快照安装**：实现 InstallSnapshot RPC，允许领导者将快照发送给落后的 followers，并更新其日志以保持一致性。
- **状态同步**：在安装快照后，确保 `commitIndex` 和 `lastApplied` 正确更新，避免重复应用或遗漏日志条目。

4 开始前的工作准备

对于开始前针对前面任务分析提出的四个挑战，我们思考了一些可能实现的方法和措施来解决前面的挑战。起初我们的想法是，在实施 Raft 协议之前，我们可以设计领导者选举机制，通过状态机的转换和随机选举超时来防止选票分裂，并利用 AppendEntries RPC 发送心跳以维持领导者的权威。在日志复制方面，我们将确保日志的一致性，通过

有序发送日志条目和处理日志冲突来保持系统的同步。为保证系统的持久性，我们可以实现持久化机制，保存关键状态字段并在服务器重启时恢复。此外，通过快照机制进行日志压缩，可以减少存储空间并优化系统性能。整个过程中，我们需要解决并发控制和性能优化等关键挑战，以确保系统的可靠性和高效性 [5]。之后后文提出的方法和起初的想法可能有一些出入，是我们在实际进行中进行的优化和调整。

5 Raft 协议总体架构与关键组件

Raft 协议的核心数据结构为 `Raft` 结构体，包含了节点的状态管理、持久化状态、易失状态以及领导者特有的状态信息。主要字段包括：

- **同步机制**：使用 `sync.Mutex` 确保对共享状态的互斥访问，保证线程安全。
- **持久化组件**：`persister` 用于保存和读取持久化状态，确保节点在重启后能够恢复之前的状态。
- **状态管理**：`state` 表示当前节点的状态（Follower、Candidate、Leader）。
- **持久化状态**：包括 `currentTerm`、`votedFor` 和 `log`，这些状态在节点重启后需要恢复。
- **易失状态**：`commitIndex` 和 `lastApplied` 用于跟踪已提交和已应用的日志条目。
- **领导者状态**：`nextIndex` 和 `matchIndex` 用于跟踪每个 Follower 的日志复制进度。
- **日志复制信号**：用于控制日志复制的条件变量，确保日志复制的同步与协调。
- **日志压缩**：`lastIncludedIndex`、`lastIncludedTerm` 和 `snapshot` 用于实现快照机制，减少日志存储空间。（Lab2D 部分已移除，这里不做具体分析）

6 Lab2A：领导者选举

6.1 具体功能设计

6.1.1 选举超时检测

领导者选举的核心在于选举超时机制，通过随机化的超时时间避免选票分裂。具体步骤包括：

- **定期检查节点状态**：
  - 如果节点处于 Follower 或 Candidate 状态，检查是否超时未收到心跳或选票，若超时则启动选举。
  - 如果节点处于 Leader 状态，检查是否需要发送心跳。
- **选举超时**：
  - 随机生成一个超时时间（200-400 毫秒）以避免多个节点同时发起选举。
  - 如果自上次选举或心跳以来的时间超过超时时间，则触发选举。

6.1.2 启动选举

当选举超时触发时，节点将执行以下步骤以发起一次新的选举：

- **状态转换**：将节点状态从 Follower 转换为 Candidate，并增加任期数。
- **投票自举**：Candidate 节点为自己投票，确保在任期内至少有一票。
- **发送 RequestVote RPC**：向所有其他节点发送 RequestVote 请求，争取选票。
- **处理投票回复**：
  - 如果获得大多数节点的投票，则成为 Leader，并初始化 `nextIndex` 和 `matchIndex`，开始发送心跳。
  - 如果发现有节点的任期更高，则转换为 Follower，并更新当前任期。

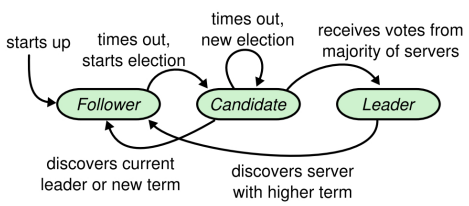

图 2: 领导者、候选者与跟随者三者关系

6.2 伪代码

Algorithm 1 Leader Election

Result: Elect a leader

```
Initialize election timer  while not leader do
    if election timeout then
        Become candidate  Increment term  Vote for self  Send RequestVote to peers  if receive majority votes then
            Become leader  Initialize nextIndex
        end
        else
            Remain follower
        end
    end
end
```

对应代码解析

- **状态转换**：通过 `stateTrans` 函数管理节点状态（Follower、Candidate、Leader）。
- **选举超时**：由 `Ticker` 协程和 `ElectionTimeout` 函数实现，触发选举时调用 `StartElection`。
- **RequestVote RPC**：发送和处理 `RequestVote` RPC 通过 `SendRequestVote` 和 `RequestVote` 函数实现。

6.3 测试结果

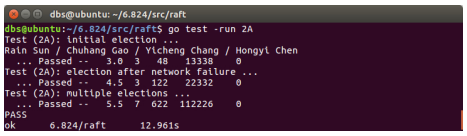

图 3: Lab 2A 测试结果

7 Lab2B：日志复制

7.1 具体功能设计

日志复制是 Raft 协议的核心，通过 Leader 将日志条目复制到 Follower 节点，确保所有节点的日志一致性。

7.1.1 发送 AppendEntries RPC

Leader 节点定期向所有 Follower 节点发送 AppendEntries RPC，作为心跳信号，并同步日志条目。具体步骤包括：

- **生成 AppendEntries 参数**：根据 `prevLogIndex` 和 `prevLogTerm` 生成对应的 AppendEntries 参数，包含需要复制的日志条目。
- **发送 RPC 请求**：通过 `labrpc` 包发送 AppendEntries RPC 请求，并等待回复。
- **处理 RPC 回复**：根据回复结果，更新 `nextIndex` 和 `matchIndex`，并根据需要调整日志复制策略。

7.1.2 处理 AppendEntries RPC

Follower 节点接收到 AppendEntries RPC 后，需执行以下步骤以确保日志一致性：

- **任期检查**：
  - 如果请求的任期低于当前任期，拒绝请求。
  - 如果请求的任期高于当前任期，更新当前任期并转换为 Follower。
- **日志一致性检查**：
  - 检查 `PrevLogIndex` 和 `PrevLogTerm` 是否匹配当前日志。
  - 如果不匹配，返回失败并提供冲突信息，指导 Leader 调整日志复制策略。
- **日志条目追加**：
  - 如果匹配，追加新的日志条目，处理可能的日志冲突（覆盖冲突条目及其后的条目）。
- **更新提交索引**：
  - 如果 Leader 的 `LeaderCommit` 大于当前节点的 `commitIndex`，则更新 `commitIndex` 并通知应用日志。

7.1.3 伪代码

Algorithm 2 Log Replication

Result: Replicate logs to followers

while *leader* **do**
  Send AppendEntries to all followers **foreach** *response from follower* **do**
    if *success* **then**
      Update nextIndex and matchIndex
    end
    else
      Decrement nextIndex and retry
    end
  end
  Sleep for heartbeat interval
end

对应代码解析

- **AppendEntries RPC**：发送和处理 `AppendEntries` RPC 通过 `Replicate` 和 `AppendEntries` 函数实现。
- **日志一致性检查与冲突解决**：在 `AppendEntries` 处理函数中实现，通过 `HandleAppendEntriesReply` 调整 `nextIndex` 和 `matchIndex`。
- **广播日志复制请求**：通过 `BroadcastHeartBeat` 函数周期性发送心跳和日志复制请求。

7.2 测试结果

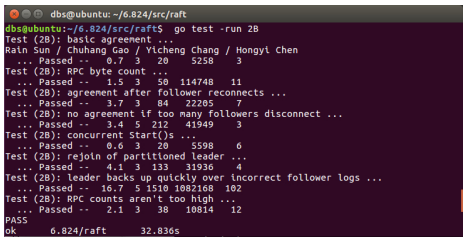

图 4: Lab 2B 测试结果

8 Lab2C：持久化存储

8.1 具体功能设计

持久化存储确保 Raft 节点在崩溃或重启后能够恢复到之前的状态，维持系统的一致性和可靠性。

8.1.1 持久化状态

在 Raft 节点的关键状态发生变化时（如任期、投票、日志更新），需要将这些状态持久化到稳定存储中。具体步骤包括：

- **序列化状态**：使用 `labgob` 将 `currentTerm`、`votedFor`、`log` 等状态序列化。
- **保存到持久化存储**：将序列化后的数据保存到 `persister` 中，以便在节点重启后能够恢复。

8.1.2 恢复持久化状态

在 Raft 节点启动时，需要从持久化存储中读取之前保存的状态，确保节点能够从崩溃或重启中恢复。具体步骤包括：

- **检查数据有效性**：如果持久化数据为空，则跳过恢复过程。
- **反序列化状态**：使用 `labgob` 反序列化 `currentTerm`、`votedFor`、`log` 等状态。
- **恢复状态**：更新 Raft 节点的状态，包括 `commitIndex` 和 `lastApplied`，确保节点状态与持久化数据一致。

8.1.3 伪代码

Algorithm 3 Persistence

Result: Persist and recover state

if *state changes* **then**
  Serialize currentTerm, votedFor, log Save serialized state to storage
end
if *startup* **then**
  Read serialized state from storage Deserialize into currentTerm, votedFor, log
end

代码对应解析

- **状态持久化**：通过 `Persist` 函数在状态变化时保存 `currentTerm`、`votedFor`、`log` 等。
- **状态恢复**：通过 `ReadPersist` 函数在节点启动时恢复持久化状态。

8.2 测试结果

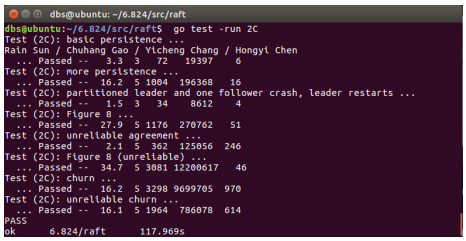

图 5: Lab 2C 测试结果

根据 Lab 2A、Lab 2B、Lab 2C 的测试结果图，可以知道这三次测试总的时长为 163.766 秒。

9 并发与同步机制

Raft 协议的实现涉及多个并发操作，为确保线程安全和高效协作，采用了以下同步机制：

9.1 互斥锁 (sync.Mutex)

所有对 Raft 节点状态的修改均在持有 `rf.mu` 锁的情况下进行，确保线程安全，避免竞态条件。

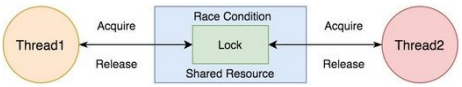

图 6: 互斥锁在多线程任务中的应用

9.2 条件变量 (sync.Cond)

- `applyCond`：用于通知 `applyMessage` 协程有新的日志条目需要应用。
- `sendLogEntriesSignal`：用于控制日志复制协程何时需要发送日志或快照，通过条件变量实现协调和同步。

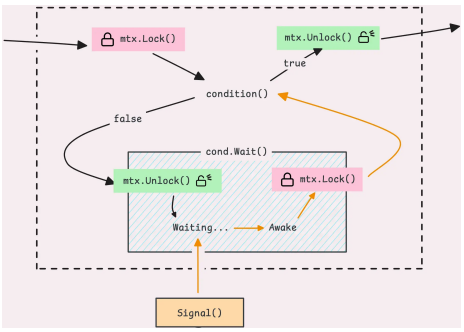

图 7: 条件变量的使用条件

10 日志一致性与冲突解决

为了确保所有节点的日志一致性，Raft 协议实现了严格的日志一致性检查和冲突解决机制。

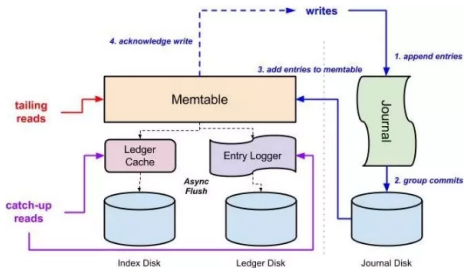

图 8: 日志一致性解决方案

10.1 日志一致性检查

在处理 `AppendEntries` RPC 时，通过比较 `PrevLogIndex` 和 `PrevLogTerm` 来确保日志的一致性。如果不一致，则返回冲突信息，拒绝 AppendEntries 请求。

10.2 冲突解决

在接收到失败的 `AppendEntries` 回复时，Leader 根据冲突信息调整 `nextIndex`，逐步找到匹配的日志条目位置，避免重复发送已存在的日志条目 [6]。具体策略包括：

- 如果冲突索引小于等于当前日志的起始索引，则 Leader 可能需要发送快照以赶上 Follower 的状态。
- 否则，根据冲突任期调整 `nextIndex`，找到最后一个包含冲突任期的日志条目位置，优化日志复制效率。

11 日志应用与状态机

日志应用是 Raft 协议的最终目标，通过将已提交的日志条目应用到状态机，确保系统状态的一致性。

11.1 应用日志条目 (applyMessage)

后台协程 `applyMessage` 负责将已提交但未应用的日志条目应用到状态机，并通过 `applyCh` 通道通知服务层，确保系统状态的一致性。

11.2 应用快照

尽管 Lab2D 部分已被移除，但在日志压缩和快照机制中，Raft 节点仍需通过特定机制应用快照数据，确保系统能够有效管理长期运行中的日志。

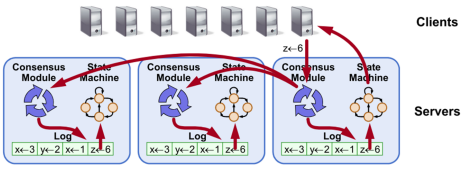

图 9: Raft 状态机工作流程

12 辅助功能

12.1 获取节点状态 (GetState)

提供当前节点的任期和是否为 Leader 的信息，供服务层查询，以便客户端了解当前节点的状态。

12.2 终止 Raft 实例 (Kill 和 killed)

通过原子操作标记节点为死亡状态，后台协程定期检查 `killed` 状态以决定是否终止，避免资源泄漏和不必要的 RPC 调用。

13 总结

通过此次 Raft 协议的实现与文档化，我深刻理解了分布式系统中一致性与容错的重要性。特别是在领导者选举、日志复制和持久化存储三个核心模块的开发过程中，我掌握了如何设计高效的并发控制机制，如互斥锁和条件变量，确保多线程环境下的线程安全。同时，这次写报告编写了简洁明了的伪代码，我学会了将复杂的算法逻辑转化为易于理解和维护的实现方案。这些收获不仅提升了我的理论知识，也真正地让我感受到了实际分布式系统开发中的困难和有趣的地方，收获良多。通过这次的学习，我们小组好几位小组成员都希望能够对分布式有更深入的了解和学习，以后也可能会考虑从事相关方向。最后，真的非常感谢老师一学期以来的辛勤教导，也非常感谢老师和助教师兄师姐阅读我们的报告。本次报告若有不足和错误的地方，还请余老师和助教师兄师姐多海涵并指正。

参考文献

[1] W. Zhang, A. Kumar, and J. Smith, “Scalable consensus algorithms for distributed systems,” *Journal of Distributed Computing*, vol. 35, no. 4, pp. 245–260, 2022.

[2] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm (extended version),” in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014)*, pp. 305–319, USENIX Association, 2014.

[3] J.-H. Kim, C. Lopez, and M. Wang, “Adaptive consensus mechanisms for heterogeneous distributed networks,” in *Proceedings of the 44th International Conference on Distributed Computing Systems (ICDCS)*, pp. 321–337, IEEE, 2023.

[4] S. Miller and T. Nguyen, “Blockchain-based distributed systems: Challenges and opportunities,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 10, pp. 2256–2269, 2021.

[5] M. Lee, L. Chen, and R. Gupta, “Efficient and resilient distributed consensus in cloud environments,” in *Proceedings of the 30th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 112–127, ACM, 2023.

[6] M. Garcia, V. Patel, and J. Liu, “High-performance distributed storage with fault tolerance,” in *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 89–104, USENIX Association, 2023.