**主题：** 分布式系统大作业-Raft 实验报告

**姓名：** 黄彬寓

**学号：** MF20330030

**日期：** 2020.12.28

# 1 分析与设计

## 1.1 step1. 设计 Raft 及四类 RPC 结构体

首先是最重要的 Raft 结构体，除了原有的锁、集群内的 serverID、自己的 ID 等数据外，根据 Raft 论文的 Figure2，我们可以把所有 servers 的持久化数据 (currentTerm、voteFor、log 数组)、易挥发数据 ( commitIndex、lastApplied)、leader 专属数据 (nextIndex 数组、matchIndex 数组) 都写入结构体中，再加上自己定义的为了更方便处理的变量和 heartbeat、election 定时器以及模拟状态机的 ApplyMsg 类型的信道。其中日志 log 是需要我们自己定义的，根据论文题意，显然放入 Index、term、command 这三项数据是合理且足够的。

接着定义 RequestVoteArgs、RequestVoteReply、AppendEntriesArgs、AppendEntriesReply 这四个结构体，其中前三个结构体按照 Fugure2 实现已经能够满足需要，而 AppendEntriesReply 中，为了能够让 nextIndex 快速回退或快速推进，需要额外定义一些变量。

## 1.2 step2. 设计 Make 与 Start，完成总体框架

首先，我先完善 Make() 与 Start() 这两个函数，因为这两个函数是通过外部接口调用的，Make() 实现一个 server 节点的各种任务，Start() 实现 client 向 server 发送 request。

在 Make() 中，首先完成对 rf 结构体的初始化，如果之前保存了持久化数据，则调用 readPersist 读取；随后是一个大的循环，在该循环内，除非触发了 Kill()，否则认为该 server 没有奔溃，持续运行，该循环体我定义为 Loop() 函数。设计 Loop() 的思路是这样的，除了 client 发出 request、收到 RequestVote 和 AppendEntries RPC 外，所有可能的操作都会在 Loop()(或被 Loop() 调用的函数) 中发生。那么显然，我们按照 server 的角色或者定时器到点这两种逻辑来设计 Loop() 是更合理的。这里 Loop() 的伪代码在图 1 中显示，我使用的是外层循环为定时器，内层循环为 rf 的角色状态。

在 election 定时器到时，根据当前状态，若为 follower 或 candidate 则开始进行选举，若为 leader 则跳过，但是考虑到这也可以是选举完刚产生的 leader，所以使用一个 bool 变量来判断是否是新当前新选举出的 leader，若是则立刻执行发送 heartbeat 通报给其他 servers。

在 heartbeat 定时器到时，根据当前状态，若为 follower 或 candidate 则重置定时器并跳出，若为 leader 则发送 heartbeat 并在内部重置 heartbeat 定时器。这里我为了加快日志复制的效率，将系统中所有的 heartbeat 也当作普通的 AppendEntries 来发送，区别仅在于这样使普通的 heartbeat 也能够附带上日志，加快复制进度。

在 Start 函数中，我们需要做的是模拟 server 收到来自某一个 client 发出的 request。那么首先判断该 server 是否是 leader，如果不是应该将该 request 转交给 leader 或者拒绝该 request，这里为了简单起见选择直接拒绝 request。随后，创建一个新日志条目项，将该 request 的 command 存入该条目，

```go
func (rf *Raft) Loop() {
    rf.ResetElectionTimer()
    rf.ResetHeartbeatTimer()
    for {
        select {
        case <-rf.electionTimer.C:
            rf.ResetElectionTimer()
            var ifLeader_NewLeader bool = false
            if rf.state == FOLLOWER_STATE {
                // times out, starts election
                ifLeader_NewLeader = true
                rf.FolCanStartElection()
            }else if rf.state == CANDIDATE_STATE { // there should not b
                ifLeader_NewLeader = true
                rf.FolCanStartElection()
            }
            if ifLeader_NewLeader == true && rf.state == LEADER_STATE {
                rf.CandidateBecomeLeader()
                //rf.LeaderSendHeartbeat()
                go rf.LeaderSendAppendEntries2()
            }
        case <-rf.heartbeatTimer.C:
            switch rf.state {
            case FOLLOWER_STATE:
                rf.ResetHeartbeatTimer()
            case CANDIDATE_STATE:
                rf.ResetHeartbeatTimer()
            case LEADER_STATE:
                go rf.LeaderSendAppendEntries2()
            }
        case <-rf.killed:
            //fmt.Println("Server ",rf.me," has crashed.")
            return
        default:
            rf.AllServerApplyLogEntries()
        }
    }
}
```

图 1: 框架函数 Loop

并将其加入 leader 的 log 中，调用 persist 完成持久化，随后再调用 LeaderSendAppendEntries 立即发送一个 AppendEntries RPC。

### 1.3 step3. 完成选主部分代码

这部分主要涉及函数 FolCanStartElection、MakeRequestVoteArgs 和接收者的 RequestVote。

MakeRequestVoteArgs 指生成一个即将作为参数被调用的 RequestVoteArgs 变量，在其内部对该值的各个成员变量进行赋值。

FolCanStartElection 指 follower 或 candidate 在 electionTimeout 定时器触发后进行的选主过程，为了确保选主的严谨这里我没有在函数在 Loop() 中被调用前添加 go 使其并发处理。在该函数中，首先修改自己的角色状态为 candidate，并将 currentTerm 加一，将 voteFor 赋值为自己的 ID。做完这些工作后，调用 MakeRequestVoteArgs 生成 RequestVoteArgs 变量，随后创建一条 RequestVoteReply 类型的信道，用于存放后续会收到的 reply。接着就是进行一个循环，对每个除自己外的 server，都并发地向对方发送 RequestVote RPC，若收到答复则添加进信道中。若在一个 heartbeat 周期内没有收到则继续发送，直到超过 electionTimeout 则停止向对方发送。将所有收到的 reply 进行统计，判断每

一个 reply 的 Term 和 VoteGranted，若存在 Term 大于自己，则变为 Follower；若 VoteGranted 为 true，则统计数加一，最后当统计数为一个多数派时，则认为自己成为该 Term 下的 leader。

RequestVote 函数中需要做的是接收者收到一个 RequestVote RPC 后进行的处理。首先判断 args 中的 Term，若小于自己的 currentTerm 则拒绝并跳出；若大于，则更新自己的角色和状态并回复支持该 candidate；若相等，则若还没有投票或者已经投给了该 candidate，则考虑下一阶段：判断该 args 中代表 candidate 最新日志项的 Index 和 Term 与自己最新日志项的 Index 和 Term，若该 candidate 的 Term 更大或 Term 相同且 Index 不小于自己最新日志项的 Index，则认为该 candidate 拥有更新的日志，选择支持它。最后需要注意的是如果选择支持它，则重置自己的 election 定时器，避免不必要的选主服务器过多造成的阻塞。

### 1.4 step4. 完成 AppendEntries 部分代码

这部分主要涉及函数 LeaderSendAppendEntries2、SendAndRevAppendEntries、MakeAppendEntriesArgs 和接收者的 AppendEntries。

MakeAppendEntriesArgs 指对指定的 server 生成 AppendEntriesArgs 变量，我们需要根据 nextIndex 来对 args 中的 PrevLogIndex 和发送哪些日志进行确定。

LeaderSendAppendEntries2 和 SendAndRevAppendEntries 是主体，代表 leader 发送 AppendEntriesArgs 并根据收到的 reply 进行操作。这里我的代码中还有 LeaderSendHeartbeat、LeaderSendAppendEntries、ReadyToSendAppendEntries，这是原来我对 AppendEntries 的处理，我起初希望和处理 RequestVote 一样，进行一个统一发送、sleep 一段时间的同时信道接收 reply、统一对 reply 进行处理，但是其实 RequestVote 中需要这样做是因为这样可以更方便地统计收到的投票数，好判断是否有一个多数派投给了自己从而确定自己能够成为 leader 或者进行下一轮选主。而处理 AppendEntries 中其实这样统一处理还使得延迟较大，因为这里我们根本不需要统一接收来做一些统一的操作。所以这几个函数被我舍弃，换成了新的版本。

在 LeaderSendAppendEntries2 中，同样是通过循环调用多个并发 go 线程向各个 server 发送 AppendEntries，而在各个线程中，我们调用函数 sendAppendEntries 并等待返回值，若返回值为 false 则认为发送失败选择丢弃；若返回值为 true，则收到了 reply，根据 reply 的内容，我们选择是否变为 follower，或者更新 nextIndex 和 matchIndex 的值。如图 2 是该函数的实现方式。

接收者的 AppendEntries 较为冗杂，为了更好地协调 leader 和 follower，我在 AppendEntriesArgs 结构体中多加了一个 LeaderSendNum 变量，因为网络中出现乱序是非常正常的，follower 很有可能收到一个过期的包，那么这个包对 follower 来说帮助并不大，甚至对其做过多处理的话会导致 log 和 leader 处的 nextIndex 等数据回退，加剧了工作量，所以通过这个 LeaderSendNum 变量来限定收到的 RPC 一定是来自 leader 的最新的 RPC。

在 AppendEntriesReply 中加入了 Contain、NextTestIndex、Newest 变量，Contain 取代了论文中 Success 的工作，代表该 follower 是否含有 prevLogIndex、preLogTerm 的日志项，原来的 Success 代表是否接受了这个 RPC。Newest 类比于 args 中的 LeaderSendNum，若 LeaderSendNum 是当前看到已知最大的，那么 Newest 就为 true，代表 follower 告诉 leader 这是我收到的最新的 RPC，那么 leader 就认为处理这个 reply 是有必要的，否则 Newest 为 false 代表这个 args 是陈旧的，那么没有处理的必要。NextTestIndex 代表 follower 告诉 leader 自己想收到从 NextTestIndex 开始的日志项。当 Contain 为 true，那么代表 PrevLogIndex 之前的都已经存入，现在想要更新的日志项且从 NextTestIndex 开

```go
func (rf *Raft)LeaderSendAppendEntries2() {
    // 对LeaderSendHeartbeat和LeaderSendAppendEntries做统一优化处理
    rf.mu.Lock()
    rf.leaderSendNum = rf.leaderSendNum + 1
    rf.ResetHeartbeatTimer()
    rf.mu.Unlock()
    for i:= 0; i< len(rf.peers); i++ {
        if i == rf.me {
            continue
        }
        if rf.state == FOLLOWER_STATE {
            break
        }
        //args := rf.MakeAppendEntriesArgs(i, false)
        go rf.SendAndRevAppendEntries(i)
    }
    rf.mu.Lock()
    rf.persist()
    rf.mu.Unlock()
    rf.LeaderCommitLogEntries()
    rf.AllServerApplyLogEntries()
}
```

图 2: LeaderSendAppendEntries 实现

始；当 Contain 为 false，代表 PrevLogIndex 处的日志项不匹配，需要回退到 NextTestIndex 开始的位置发送新的日志项。基于 NextTestIndex 这个变量，我们就能够实现一次性回退多个的思想，提升了效率。

### 1.5 step5. 一些琐碎的工作

LeaderCommitLogEntries 指 leader 通过 matchIndex 数组更新自己的 commitIndex，该函数会在每次 leader 发送 AppendEntries 后被调用一次进行更新。这里我进行了比较小的优化，按照论文的意思，对当前 commitIndex+1 开始，若有一个多数派的 matchIndex 不小于该数，那么这个 Index 的日志项是可被提交的，这里我使用一个数组来统计每个 Index 下 match 的 server 数量，这样可以节约很多比较产生的时间。这里需要注意的是 leader 更新 commitIndex 后会把自己最新的 commitIndex 在 AppendEntries RPC 中携带发给 follower，follower 根据该项更新自己的 commitIndex，但是需要注意不能超过自己日志长度，即在 args.LeaderCommit 和 len(rf.log) 取一个较小值。

AllServerApplyLogEntries 指当 commitIndex 大于 lastApplied 时，可以把从 lastApplied+1 到 commitIndex 的日志项 apply 到自动机中，在这里代表传入 ApplyMsg 类型的信道中。为了使得 lastApplied 尽量和 commitIndex 同步，除了在 AppendEntries 的发送方和接收方函数尾部添加该函数外，在 Loop() 函数中，当 select 选择 default 时也自动执行 AllServerApplyLogEntries。

persist 和 readPersist 用于数据持久化，使得部分重要的数据在服务器 crash 之后的恢复过程中能够从 stable storage 中读取这部分重要数据，从而更快地恢复成集群中的一员而不使得 leader 发来的日志项过多导致网络负担较大。这里我选择持久化的数据有 currentTerm、voteFor、electionTimeOut、log。以及在各个函数中当上述变量发生更改时会在后续中调用 persist 更新数据。

基于上面在 AppendEntries 两个结构体中多加的变量用于提升日志复制效率，在 Raft 节点中加入了一个新的 Boolean 型变量 consistent，仅供同步阶段的 follower 使用，当 leader 发来的 RPC 中 follower 含有对应 PrevLogIndex、PrevLogTerm 处的日志项时，reply 中的 Contain 设为 true，并导

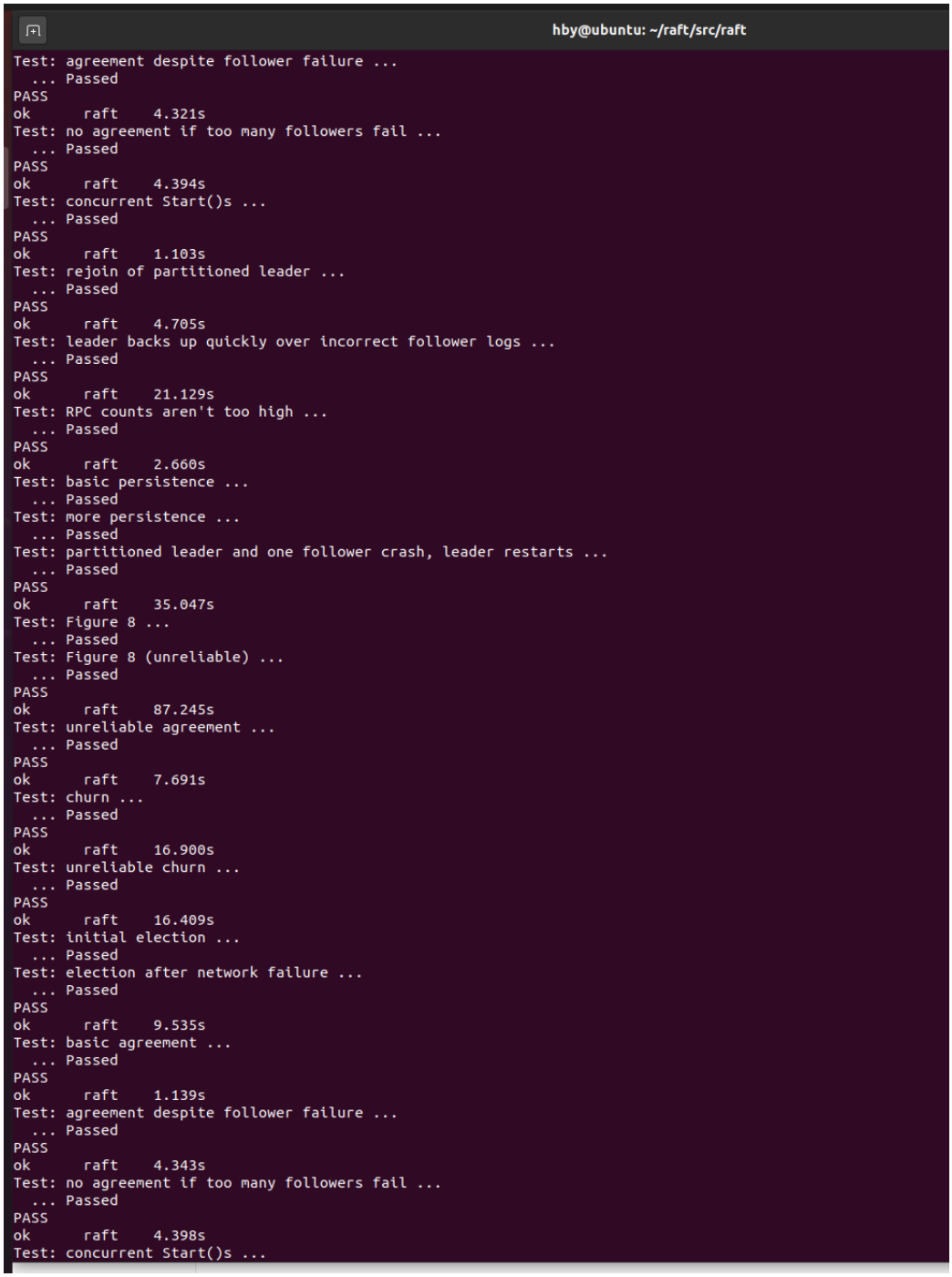
致自己的变量 consistent 也变为 true，代表自己已经不需要回退了，这样下次再收到 RPC 时根据自己的 consistent 为 true，可以直接开始加入日志项而不用考虑回退 nextIndex 的问题。但是这会带来另一个问题，就是当 leader crash 后，新的 leader 发来 RPC 这时候和新的 leader 的日志并不一定是从 nextIndex 开始匹配的，所以当发现 voteFor 不等于收到的 RPC 中的 LeaderId 或 RPC 中的 Term 比自己的 currentTerm 大时，就重置 consistent 为 false。

## 2 完成情况

Part1、2、3 必做和选做部分均已完成，但是 Persist2、Figure8Unreliable 的通过率大概在一半左右，UnreliableChurn 也大概 10 次会有一次 fail，其他用例没有发现错误。疑似原因是这样的：

- 实验有一定延迟，Persist2 中错误的那些情况经打印数据发现基本是 leader 还没来得及将最新的那一个日志项发送给所有 server，部分 server 就被关闭，导致最后报出 failed to reach agreement 的错误提示。
- 怀疑 AppendEntries 后来由于优化性能加入了一些新定义的参数后，有可能也把 bug 带进来了，还需要继续优化。

实验演示如图 3，这是我编写了一个简单脚本循环测试程序 20 次的截图。(有时测试中会看到 Persist2 和 Figure8Unreliable 失败的情况)




# 3 总结

## 3.1 一路上遇到的困难

这次大实验一路上遇到了很多 bug。

最开始，卡在了 TestReElection，检查到错误原因是 server2 断连后，server0 与 server1 循环选举，并判断到对方的 term 更大而转为 follower，从此往复。后来找到原因是在 FolCanStartElection 中，

reply.Term > rf.currentTerm 写成了 rf.me。

后来，卡在 TestBasicAgree，检查到错误原因是刚开始各自选主发送 heartbeat 等正常，后来变成 server1 与 server3 进入 candidate 且死循环，从此往复。很明显，server0、2、4 在这里没有 crash 或延迟，测试后发现 server2 成为 leader 后发 heartbeat+AppendEntries+heartbeat 后，在第二个 heartbeat 处卡死，最终锁定问题：leader 在发 AppendEntries 后 sleep 一段时间内下一个 heartbeat 触发 leader 执行 heartbeat，此时还没经历 AppendEntries 的 LeaderCommitLogEntries，所以发出去的 prevLogIndex、prevLogIndex 是过期的，后来使 nextIndex=matchIndex=0。

再后来，依然卡在 TestBasicAgree，错误原因：commitIndex 与 matchIndex 问题，follower 端的 commitIndex 和 LastApplied 迟迟不动。后来找到原因，Index 时从 1 开始的，所有与日志项有关的包括 index、commitIndex 等, 现在开始给所有需要偏移 1 的变量进行偏移，在这一步上我修改了很多处很细节的地方，花了不少时间。

再后来，卡在 Persist2，有不少报错原因是 panic:send on closed channel，这让我想到是在 RequestVote 处理中，由于某一部分测试设置的传输延迟较大，导致我关闭了信道后还有部分 reply 朝信道中发送导致 panic，于是我使用互斥信号量和一个 Boolean 来约束当信道关闭后，无法再有 reply 朝信道中传输。

### 3.2 后续工作

由于我的工作和 Raft 有关系，所以能够完善自己写的这份代码是非常有必要的，首先我需要优化 AppendEntries 函数，在不降低性能的情况下将里面的逻辑写的更清晰，其次我需要优化 Loop() 函数，因为 for 循环内的最外层循环使用角色来判断进入哪一个分支无疑是更符合逻辑的，而且也能够省去部分定时器的开销。

### 3.3 致谢

这次大实验让我对共识协议有了更深的认识，尽管我原来的工作就是共识协议，但是由于刚进组，对共识协议如 Paxos、Raft、ZAB 等都是停留在纸面上和规约证明上，这次从 code 的角度让我对 Raft 有了更深的认识，让我发现我以前看论文的时候以为自己读懂了，但其实有很多细节的地方也并没有弄得很清楚。最后，感谢老师和助教这一学期的工作！