

考核项目

基于raft协议的简易Replicated Key Value Store

徐宇鸣

2020.2.21

该程序主要实现要点如下：

- 1、基于所需功能，我认为这个考核项目的实现需要高并发，因此选择了实现并发相对轻松容易的golang语言来进行编写
- 2、在经过了调研之后，我决定把重点放在容错协议raft上，因此所实现的kvstore仅仅是一个简单的map[string]string，log也是选择使用放在内存中的简单数组
- 3、在使用rpc方面选择了grpc，这也是我选择golang语言的原因之一，因为在golang语言环境下grpc的使用也十分方便
- 4、在并发方面由于本次考核项目中存在如需要在不同条件下修改的计时器等变量，critical area无法避开，在考虑完之后决定不纠结于性能而将所有的critical area全部上锁

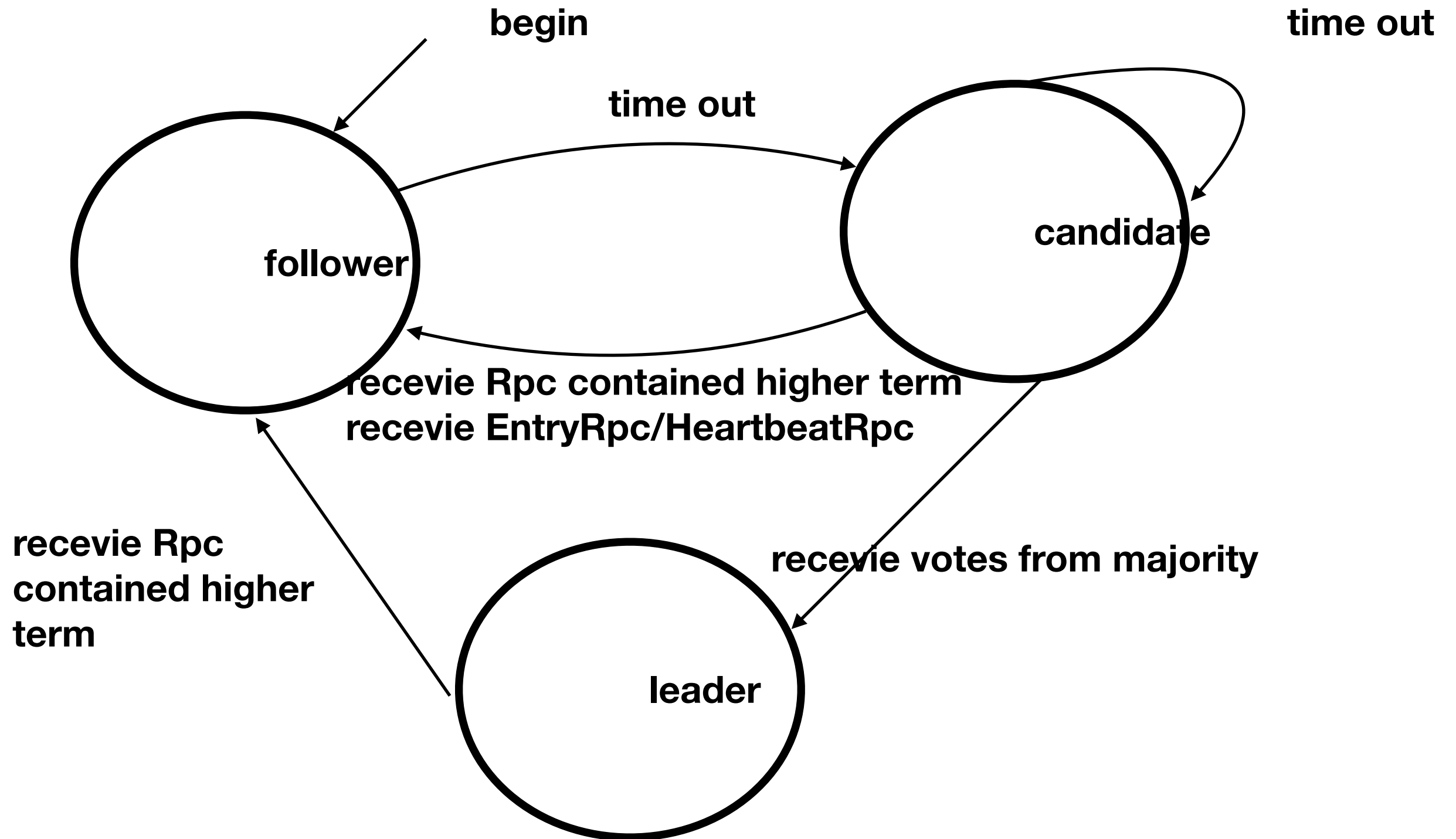
```
simple_kvstore map[string]string //need to be  
simple_raftlog_key [MAXLOGLENGTH] string  
simple_raftlog_Value [MAXLOGLENGTH] string  
simple_raftlog_Term [MAXLOGLENGTH] uint64  
simple_raftlog_length uint64
```

简易的kvstore和log

```
//LOCK  
var timerLock sync.Mutex // for time  
var serverLock sync.RWMutex // for state  
var logLock sync.RWMutex // for log  
var IndexLock sync.RWMutex // for nextIndex and MarchInd
```

所有使用的锁

server之间的状态转换：



实现的server逻辑上的一些重要函数线程

all servers:

Commitlog

每1ms检查一次commitIndex
并进行kvstore更新

Server(lis)

grpc的服务调用函数，接受rpc
请求

TransMsg

服务器之前的rpc响应函数，也
是本次实现的重点

MaintainState()

类似于状态机，根据state进行
操作

RecvCmd

用户与服务器rpc的响应函数，
也是本次实现的重点

TimerFunc

用来检测计时器的状态从而变
更服务器的状态

所使用的grpc中交互的信息：

```
message Request{
    string name = 1;
    string value = 2;
}

message SuccessMsg{
    bool success = 1;
}

message Entry {
    EntryType type = 1;
    uint64 term = 2; //log entry term
    uint64 index = 3; //log entry index
    string key = 4; //stored data
    string value = 5;
}

message MessageSend{
    MessageSendType type = 1;
    //0 stands for AppendEntries RPC, 1 stands for RequestVote
    //2 stands for heartbeat
    uint64 term = 2;
    //leader's or candidate's term
    uint64 logIndex = 3;
    //app: prelogindex, vote: lastlogindex
    uint64 logTerm = 4;
    //the same
    uint64 commitIndex = 5;
    //commit index
    uint64 nodeID = 6;
    //leaderID or candidateID
    Entry entry = 7;
    //entries[], only set one
}

message MessageRet{
    MessageReturn type = 1;
    uint64 term = 2;
    uint64 success = 3;
    //app:success, vote:voteGranted
}
```

这里我因为把重点放在了raft上，只是简单地让用户传入了需要我们插入的key-value

由于RequestvoteRpc和AppendEntryRpc的结构非常相近（只差一个entry[]，在本次编写中我只选取了一个entry），我们只需要加入一个Type进行辨认即可

TransMsg函数与论文中Figure2部分对比：

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
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

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

```
default://heartbeat or appendEntry RPC
if msgSend.Term < s.currentTerm { 对应1
    //log.Println("out-of-date,reject")
    msgRet.Success = 0
}else {
    ResetClock() 当Term检查有效后不管成功与否我们都得重置时钟
    //if msgSend.Type!=2{
    //    //log.Println("oh it is here:111")
    //}
    s.CheckSendIndex(msgSend.CommitIndex) 对应5
    //log.Println(msgSend.CommitIndex,s.CommitIndex,s.simple_raftlog_length)
    s.leaderId = msgSend.NodeID
    //log.Println("lock check")
    logLock.RLock() 对应2
    //log.Println("msg check")
    if s.simple_raftlog_length < msgSend.LogIndex {
        //didn't exist a Entry in the previous one
        //log.Println("too advanced")
        logLock.RUnlock()
        msgRet.Success = 0 对应3
    }else if s.simple_raftlog_length >= msgSend.LogIndex &&
    msgSend.LogTerm != s.simple_raftlog_Term[msgSend.LogIndex]{
        //there is a conflict,delete all entries in Index and after it,
        //and we can't append this entry
        //log.Println("conflict,reject")
        logLock.RUnlock()
        logLock.Lock()
        s.simple_raftlog_length = msgSend.LogIndex - 1
        logLock.Unlock()
        msgRet.Success = 0
    }else{
        if msgSend.Type!=2{
            //log.Println("oh it is here:133")
        }
        msgRet.Success = 1//actually msgSend.LogIndex = msgSend.Entry.Index - 1
        if msgSend.Type != 2 &&
        s.simple_raftlog_length >= msgSend.Entry.Index &&
        msgSend.Entry.Term != s.simple_raftlog_Term[msgSend.Entry.Index]{
            //unmatch the new one,conflict delete all, then append
            //log.Println("unmatch the new one,delete and then append")
            logLock.RUnlock()
        }
```

在任何情况下，当follower收到一个term大于或等于currentTerm的Rpc时，都是有效的，都需要进行处理（重置时钟）

TransMsg函数与论文中Figure2部分对比：

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
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

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

```
logLock.RUnlock()
logLock.Lock()
s.simple_raftlog_length = msgSend.LogIndex
logLock.Unlock()
s.AddEntry(msgSend.Entry.Key, msgSend.Entry.Value, msgSend.Entry.Term)
}else if msgSend.Type != 2 &&
s.simple_raftlog_length >= msgSend.Entry.Index &&
msgSend.Entry.Term == s.simple_raftlog_Term[msgSend.Entry.Index]{
    //log.Println("not the new one, don't have to append")
    logLock.RUnlock() //not the new entry, and we don't need to delete
}else if msgSend.Type != 2 && s.simple_raftlog_length == msgSend.LogIndex {
    //log.Println("yes it is a new one, append")
    logLock.RUnlock() //new one, and there is not conflict
    s.AddEntry(msgSend.Entry.Key, msgSend.Entry.Value, msgSend.Entry.Term)
}else { //it is heartbeat
    if msgSend.Type != 2 {
        //log.Println("bug: it is not heartbeat!!")
    }
    logLock.RUnlock()
}
```

```
//if msgSend.Type!=2{
//    log.Println("oh it is here:155")
//}
```

就如论文中所提到的，我们对于entry的判断是与其index以及term进行判断，不对内容进行检查，当两个entry有着一样的index以及term，我们就认为他们是一样的

TransMsg函数与论文中Figure2部分对比:

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

```
msgRet.Term = s.currentTerm
switch msgSend.Type {
case 1://voteRPC
    serverLock.RLock()
    if msgSend.Term < s.currentTerm {           对应1
        serverLock.RUnlock()
        log.Println("node:",s.NodeID,"lower term, reject",msgSend.NodeID)
        msgRet.Success = 0
    } else {
        if s.votedFor == 0 || s.votedFor == msgSend.NodeID{
            serverLock.RUnlock()
            if s.IsUpToDate(msgSend.LogTerm,msgSend.LogIndex){
                serverLock.Lock()
                s.votedFor = msgSend.NodeID           对应2
                serverLock.Unlock()
                msgRet.Success = 1
            }else{
                log.Println("node:",s.NodeID,"out-of-date, reject",msgSend.NodeID)
                msgRet.Success = 0
            }
        }
    }
} else {
    serverLock.RUnlock()
    log.Println("node:",s.NodeID,"have votedFor other, reject",msgSend.NodeID)
    msgRet.Success = 0
}
}
```

关于up-to-date, 我在查阅资料之后得出以下定义:

```
func (s *server)IsUpToDate(Term,Index uint64)bool{
    logLock.RLock()
    var uptodate bool = s.simple_raftlog_length==0 ||
    Term > s.simple_raftlog_Term[s.simple_raftlog_length] ||
    (Term == s.simple_raftlog_Term[s.simple_raftlog_length] && Index >= s.simple_raftlog_length)
    logLock.RUnlock()
    return uptodate
}
```

因为我的数组从index=1开始使用, 这里怕报错就写了这个

先考虑term

再考虑index的长度


```

func (s *server)IsUpToDate(Term,Index uint64)bool{
    logLock.RLock()
    var uptodate bool = s.simple_raftlog_length==0 ||
    Term > s.simple_raftlog_Term[s.simple_raftlog_length] ||
    (Term == s.simple_raftlog_Term[s.simple_raftlog_length] && Index >= s.simple_raftlog_length)
    logLock.RUnlock()
    return uptodate
}

```

关于为什么先比较term，再比较长度的原因：

在理解之后比较可靠的是，首先lastTerm大于我们（当前收到RPC的节点）的log的lastTerm（在这里包括在写AppendEntryRpc时有一个其实本质是一样的疑问，为什么lastTerm大就可以，而不是优先考虑log的长度（index）），如果等于的情况下再比较log长度

关于疑问：如果存在一个lastTerm大，但是log长度却短于其他服务器的服务器，那么这个log一定是在这个服务器作为leader的时候创建的，而他又不可能成为leader（因为他没有up-to-date），因此矛盾

其他方面：

乱序信息的处理：

```
switch Ret.Type{
    case 0:
        IndexLock.Lock()
        if Ret.Success == 0{
            //fail to add
            //since we know that the entry can't be added
            //we should update according to LogIndex
            //it is always true that LogIndex <= s.nextIndex[NodeID]-1
            //so we don't have to set a condition
            s.nextIndex[NodeID] = LogIndex
        }else if Ret.Success == 1{//successfully add an entry
            //since we know that the entry is successfully added in the position of LogIndex
            //we should update according to LogIndex
            if LogIndex >= s.nextIndex[NodeID]-1{
                s.matchIndex[NodeID] = LogIndex+1
                s.nextIndex[NodeID] = LogIndex+2
            }
        }
        IndexLock.Unlock()
    case 1:
        if Ret.Success == 1{
            log.Println("Recieve vote from",NodeID)
            serverLock.Lock()
            s.votes = s.votes + 1
            serverLock.Unlock()
        }
    case 2://heartbeat
        IndexLock.Lock()
        if Ret.Success == 0{
            //it is always true that LogIndex <= s.nextIndex[NodeID]-1
            //so we don't have to set a condition
            s.nextIndex[NodeID] = LogIndex
        }else{
            if LogIndex >= s.nextIndex[NodeID]-1{
                //since we know that it is matched at logindex
                s.matchIndex[NodeID] = LogIndex
                s.nextIndex[NodeID] = LogIndex+1
            }
        }
        IndexLock.Unlock()
}
```

处理方式：除了我们所收到的回复信息外，我们还需要我们发出的信息中的index，这样能够帮我们定位这个信息的位置

最初我的信息传递方式是由leader的一个线程每30秒传递一次信息，而这样的做法导致commit以及log更新太慢，于是我修改了做法，收到用户调用的时候马上发送Rpc，这样的并发会使得我们收到的信息是乱序的

其他方面:

这个timer必定是全局/或作为server的attr的, 因为
我们需要在不同的条件下去改变他 (重置/停止)

clock的使用:

```
}
func (s *server)TimerFunc(){
    for{
        select{
            case <- timer.C:
                log.Println("node:",s.NodeID,":Time out, begin election")
                s.BecomeCandidate()
        }
    }
}
```

计时器的时间我选择了论文中提到的
100ms-500ms

TimerFunc线程生命周期为
整个server的生命周期

```
logLock.Unlock()
}
func (s *server)BecomeCandidate(){
    log.Println("node:",s.NodeID,":BecomeCandidate")
    serverLock.Lock()
    s.state = 2
    s.leaderId = 0
    s.currentTerm = s.currentTerm + 1
    s.votedFor = s.NodeID
    s.votes = 1;//vote for himself
    serverLock.Unlock()
    //reset timer
    ResetClock()
    s.BroadcastVoteRpc()
}
```

```
}
func ResetClock(){
    //log.Println("Clock reset")
    timerLock.Lock()
    timer.Reset(time.Duration(rand.Intn(400)+100)*time.Millisecond)
    timerLock.Unlock()
}
func StopClock(){
    //log.Println("Clock stoped")
    timerLock.Lock()
    timer.Stop()
    timerLock.Unlock()
}
```

```
}
func (s *server)ResetServer(){
    s.NodeID = ID
    s.leaderId = 0 //doesn't have a leader
    s.currentTerm = 1
    s.votedFor = 0 //doesn't vote for any one
    s.lastApplied = 0//doesn't apply anything
    s.CommitIndex = 0
    s.simple_kvstore = make(map[string]string)
    s.simple_raftlog_length = 0
    s.votes = 0
    s.state = 1//follower
    for i,serverAddress := range peers{
        if uint64(i+1) == s.NodeID{
            var blank pb.MsgRpcClient
            conns = append(conns,nil)
            clients = append(clients,blank)
            continue
        }
        conn, _ := grpc.Dial(serverAddress,grpc.WithInsecure())
        conns = append(conns,conn)
        client := pb.NewMsgRpcClient(conn)
        clients = append(clients,client)
    }
    timer = time.NewTimer(time.Duration(rand.Intn(400)+100)*time.Millisecond)
    log.Println("node:",s.NodeID,":Resetting Server")
    go s.TimerFunc()
}
func (s *server)ForwardCommand(Req *pb.Request){
```

其他方面：
处理用户请求

当我们接受到用户请求时，检查leaderId并转发，如果此时没有leader，则返回调用失败

```
func (s *server)RecvCmd(ctx context.Context,cmd *pb.Request)(*pb.SuccessMsg,error){
    start:=time.Now().UnixNano()
    //log.Println("node:",s.NodeID,":Reciving cmd from client")
    var RetMsg pb.SuccessMsg
    var index uint64
    RetMsg.Success = true
    serverLock.RLock()
    if s.leaderId !=0 &&s.leaderId != s.NodeID{//leader is not me
        serverLock.RUnlock()
        return s.ForwardCommand(cmd),nil 转发给leader
    }else if s.leaderId == s.NodeID{
        serverLock.RUnlock()
        index = s.AddEntry(cmd.Name,cmd.Value,s.currentTerm)
        s.BroadcastEntryRpc()
    }else
    {
        //当还没有leader时候需要等待，这里没有考虑错误处理（超时等）
        for s.leaderId==0{
            serverLock.RUnlock()
            time.Sleep(1*time.Millisecond)
            serverLock.RLock()
        }
        serverLock.RUnlock()
        return s.ForwardCommand(cmd),nil
    }
    serverLock.RLock()
    //无限休眠+等待，直到commit为止
    for index>s.CommitIndex{
        serverLock.RUnlock()
        time.Sleep(1*time.Millisecond)
        serverLock.RLock()
    }
    serverLock.RUnlock()
    count++
    end:=time.Now().UnixNano()
    RecvCmdTime += (end - start)
    return &RetMsg,nil
}
```

其他方面：
grpc的使用

最开始我选择使用短连接（连接后调用一次马上关闭）的方式，后面发现这样非常占用资源，于是修改成了长连接（存入存根，注册后的client也保留）的方式

```
}
func (s *server)ResetServer(){
    s.NodeID = ID
    s.leaderId = 0 //doesn't have a leader
    s.currentTerm = 1
    s.votedFor = 0 //doesn't vote for any one
    s.lastApplied = 0//doesn't apply anything
    s.CommitIndex = 0
    s.simple_kvstore = make(map[string]string)
    s.simple_raftlog_length = 0
    s.votes = 0
    s.state = 1//follower
    for i,serverAddress := range peers{
        if uint64(i+1) == s.NodeID{
            var blank pb.MsgRpcClient
            conns = append(conns,nil)
            clients = append(clients,blank)
            continue
        }
        conn, _ := grpc.Dial(serverAddress,grpc.WithInsecure())
        conns = append(conns,conn)
        client := pb.NewMsgRpcClient(conn)
        clients = append(clients,client)
    }
    timer = time.NewTimer(time.Duration(rand.Intn(400)+100)*time.Millisecond)
    log.Println("node:",s.NodeID,":Resetting Server")
    go s.TimerFunc()
}
func (s *server)ForwardCommand(Req *pb.Request){
```


简单的一些测试:

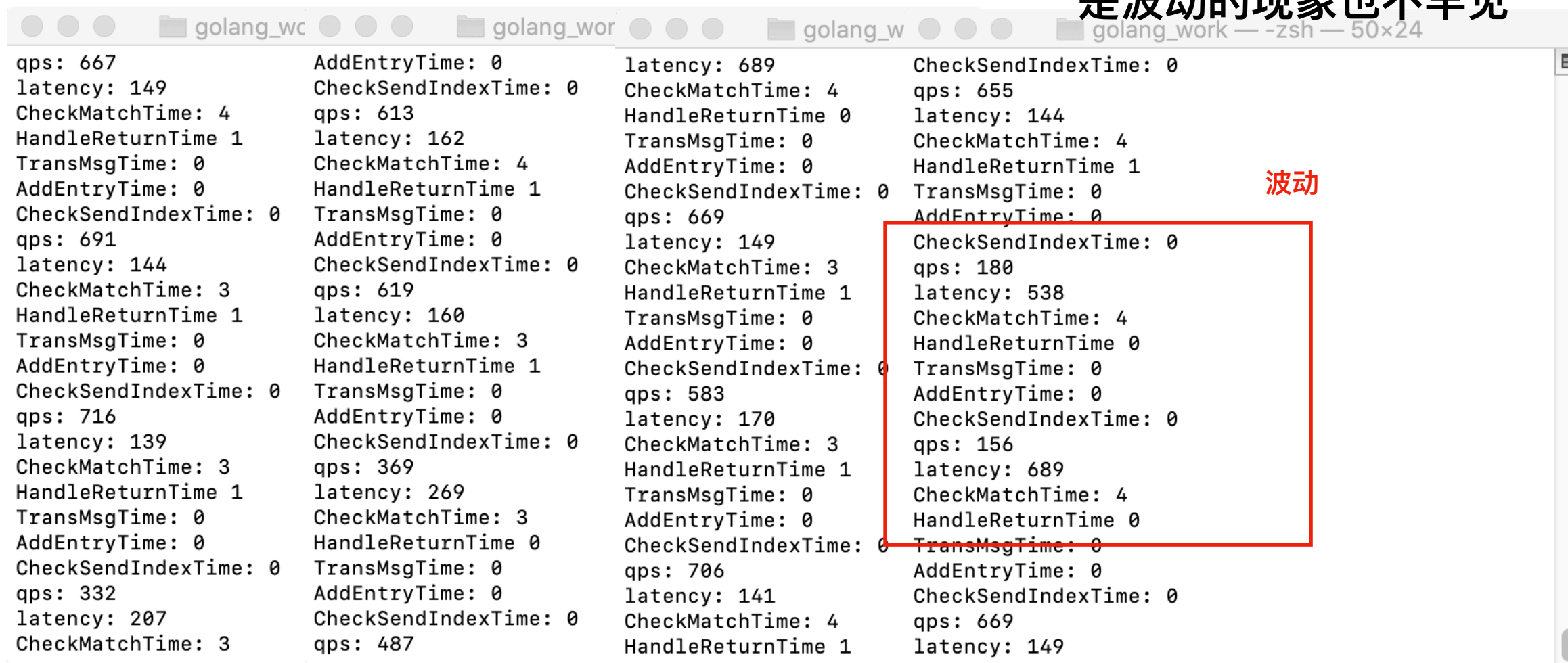
由于我本身大学以来没接触过多少测试程序的机会, 因此这里做的可能会不完善

吞吐量与延迟:

采用的方法是使用一个变量count记录客户调用次数, 每过去一秒与前一秒的记录进行相减而commit的记录方式则是运行一个线程函数, 每5秒显示一次信息

测试方式是由100个client对其进行不断发送调用

可以看到基本的qps在600-700左右, latency基本在140-200, 但是波动的现象也不罕见



The image shows a terminal window with four columns of test data. The data includes metrics like qps, latency, and various timing values. A red box highlights a section of the data where qps drops significantly from 669 to 180, with a corresponding increase in latency from 144 to 538. The Chinese word '波动' (fluctuation) is written in red next to this section.

Column 1	Column 2	Column 3	Column 4
qps: 667	AddEntryTime: 0	latency: 689	CheckSendIndexTime: 0
latency: 149	CheckSendIndexTime: 0	CheckMatchTime: 4	qps: 655
CheckMatchTime: 4	qps: 613	HandleReturnTime 0	latency: 144
HandleReturnTime 1	latency: 162	TransMsgTime: 0	CheckMatchTime: 4
TransMsgTime: 0	CheckMatchTime: 4	AddEntryTime: 0	HandleReturnTime 1
AddEntryTime: 0	HandleReturnTime 1	CheckSendIndexTime: 0	TransMsgTime: 0
CheckSendIndexTime: 0	TransMsgTime: 0	qps: 669	AddEntryTime: 0
qps: 691	AddEntryTime: 0	latency: 149	CheckSendIndexTime: 0
latency: 144	CheckSendIndexTime: 0	CheckMatchTime: 3	qps: 180
CheckMatchTime: 3	qps: 619	HandleReturnTime 1	latency: 538
HandleReturnTime 1	latency: 160	TransMsgTime: 0	CheckMatchTime: 4
TransMsgTime: 0	CheckMatchTime: 3	AddEntryTime: 0	HandleReturnTime 0
AddEntryTime: 0	HandleReturnTime 1	CheckSendIndexTime: 0	TransMsgTime: 0
CheckSendIndexTime: 0	TransMsgTime: 0	qps: 583	AddEntryTime: 0
qps: 716	AddEntryTime: 0	latency: 170	CheckSendIndexTime: 0
latency: 139	CheckSendIndexTime: 0	CheckMatchTime: 3	qps: 156
CheckMatchTime: 3	qps: 369	HandleReturnTime 1	latency: 689
HandleReturnTime 1	latency: 269	TransMsgTime: 0	CheckMatchTime: 4
TransMsgTime: 0	CheckMatchTime: 3	AddEntryTime: 0	HandleReturnTime 0
AddEntryTime: 0	HandleReturnTime 0	CheckSendIndexTime: 0	TransMsgTime: 0
CheckSendIndexTime: 0	TransMsgTime: 0	qps: 706	AddEntryTime: 0
qps: 332	AddEntryTime: 0	latency: 141	CheckSendIndexTime: 0
latency: 207	CheckSendIndexTime: 0	CheckMatchTime: 4	qps: 669
CheckMatchTime: 3	qps: 487	HandleReturnTime 1	latency: 149

容错：

可以看到当我停掉server 3 的时候，5最先超时，然后成为了leader

```
golang_work — -zsh — 53x24
nextindex: 0 0 0 0
matchindex: 0 0 0 0
commitindex: 2236 loglength: 2237
leaderID 3 lastApplied 2236
nextindex: 0 0 0 0
matchindex: 0 0 0 0
commitindex: 2388 loglength: 2389
leaderID 3 lastApplied 2387
nextindex: 0 0 0 0
matchindex: 0 0 0 0
2020/02/20 21:00:32 node: 4 :BecomeFollower
commitindex: 2482 loglength: 2482
leaderID 5 lastApplied 2482
nextindex: 0 0 0 0
matchindex: 0 0 0 0
2020/02/20 21:00:37 node: 4 :BecomeFollower
2020/02/20 21:00:37 node: 4 :Time out, begin election
2020/02/20 21:00:37 node: 4 :BecomeCandidate
2020/02/20 21:00:37 Recivie vote from 2
2020/02/20 21:00:38 node: 4 :BecomeFollower
2020/02/20 21:00:38 node: 4 :Time out, begin election
2020/02/20 21:00:38 node: 4 :BecomeCandidate
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work %

golang_work — -zsh — 51x24
nextindex: 1580 1580 0 1580 1580
matchindex: 1579 1579 0 1579 1579
commitindex: 1729 loglength: 20000
leaderID 3 lastApplied 1729
nextindex: 1731 1731 0 1731 1731
matchindex: 1730 1730 0 1730 1730
commitindex: 1881 loglength: 20000
leaderID 3 lastApplied 1881
nextindex: 1883 1883 0 1883 1883
matchindex: 1882 1882 0 1882 1882
commitindex: 2032 loglength: 20000
leaderID 3 lastApplied 2032
nextindex: 2034 2034 0 2034 2034
matchindex: 2033 2033 0 2033 2033
commitindex: 2182 loglength: 20000
leaderID 3 lastApplied 2182
nextindex: 2184 2184 0 2184 2184
matchindex: 2183 2183 0 2183 2183
commitindex: 2334 loglength: 20000
leaderID 3 lastApplied 2334
nextindex: 2336 2336 0 2336 2336
matchindex: 2335 2335 0 2335 2335
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work %

golang_work — -zsh — 50x24
leaderID 3 lastApplied 2110
nextindex: 1 1 1 1 0
matchindex: 0 0 0 0 0
commitindex: 2262 loglength: 2263
leaderID 3 lastApplied 2262
nextindex: 1 1 1 1 0
matchindex: 0 0 0 0 0
commitindex: 2414 loglength: 2415
leaderID 3 lastApplied 2414
nextindex: 1 1 1 1 0
matchindex: 0 0 0 0 0
2020/02/20 21:00:32 node: 5 :Time out, begin elect
ion
2020/02/20 21:00:32 node: 5 :BecomeCandidate
2020/02/20 21:00:32 Recivie vote from 1
2020/02/20 21:00:32 Recivie vote from 2
2020/02/20 21:00:32 Recivie vote from 4
2020/02/20 21:00:32 node: 5 :BecomeLeader
commitindex: 2482 loglength: 2482
leaderID 5 lastApplied 2482
nextindex: 2483 2483 2483 2483 0
matchindex: 2482 2482 0 2482 0
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work %

golang_work — -zsh — 52x24
leaderID 3 lastApplied 2156
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
commitindex: 2307 loglength: 2308
leaderID 3 lastApplied 2306
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
commitindex: 2458 loglength: 2459
leaderID 3 lastApplied 2458
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
2020/02/20 21:00:32 node: 2 :BecomeFollower
commitindex: 2482 loglength: 2482
leaderID 5 lastApplied 2482
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
2020/02/20 21:00:37 node: 2 :BecomeFollower
2020/02/20 21:00:37 node: 2 :BecomeFollower
2020/02/20 21:00:38 node: 2 :Time out, begin electio
n
2020/02/20 21:00:38 node: 2 :BecomeCandidate
2020/02/20 21:00:38 Recivie vote from 4
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work %

golang_work — -zsh — 51x24
leaderID 3 lastApplied 2167
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
commitindex: 2318 loglength: 2319
leaderID 3 lastApplied 2318
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
commitindex: 2470 loglength: 2471
leaderID 3 lastApplied 2470
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
2020/02/20 21:00:32 node: 1 :BecomeFollower
commitindex: 2482 loglength: 2482
leaderID 5 lastApplied 2482
nextindex: 0 0 0 0 0
matchindex: 0 0 0 0 0
2020/02/20 21:00:37 node: 1 :Time out, begin electi
on
2020/02/20 21:00:37 node: 1 :BecomeCandidate
2020/02/20 21:00:37 Recivie vote from 2
2020/02/20 21:00:37 Recivie vote from 4
2020/02/20 21:00:37 node: 1 :BecomeLeader
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work %

golang_work — -zsh — 80x
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
# command-line-arguments
./simple_test_client.go:22:12: undefined: client
./simple_test_client.go:38:21: undefined: s
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
# command-line-arguments
./simple_test_client.go:37:6: i declared and not used
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
[xuyuming@xuyumingdeMacBook-Pro-51 golang_work % go run
^Csignal: interrupt
xuyuming@xuyumingdeMacBook-Pro-51 golang_work % 3
```

考核项目总结：

实现了：

- 1、raft协议中至关重要的rpc部分，能够使5个节点的cluster在一段时间内保持相对稳定**
- 2、能够接受client的调用请求，且follower能够将client的请求转发至leader处**
- 3、最多允许2个节点发生错误，且在错误发生后cluster中的节点能够迅速保持稳定，这也是raft中要求的**
- 4、在修改了AppendEntryRpc机制（原先实现的是由leader的一个线程每30ms传递一次，这样的做法是有序的）使其收到用户调用请求马上发送Rpc之后（这样的做法是乱序的，发出以及收回的信息顺序不一定一致），能够保持信息不出错**

不足之处：

- 1、没有实现cluster membership change（config的修改）以及数据持久化（snapshot）**
- 2、代码结构方面仍然有很多不足，事实上我最开始编写的时候由于太过偷懒有很多地方实现下来使得自己debug环节吃了不少苦头；**
- 3、自己对grpc也是属于初次使用，踩了很多坑（不如说这也是自己为数不多的并发+存在通信的编程），如自己一开始包括服务器之间的通信都是使用短连接的方式而非存下存根，注册了大量的client导致测试时很容易报错**

自己的一些感触：

其实最开始接到这个考核我内心是很虚的，因为我既没多少次接触过并发也没有几次socket编程经验，对所有涉及的东西的了解可以说近乎为0，但是自己上大学以来也已经碰到过好多第一次，也坚信只要自己能将困难看清，一步一步慢慢地摸索出方法，也是能完成的，最后能做成这样我也挺满意的了。

在这次编写之后我对复现论文也有了一些新的看法，最开始我一直认为复现论文就是完全按着论文中的描写就能完成，在这次编写后才发现了如论文作者可能会为了更加容易让人弄懂而简化协议，因此其中的一些部分是可以由读者根据自身需求去定制。

在阅读论文编写程序后对raft协议也有了一定的了解，这的确是个比较好懂的协议