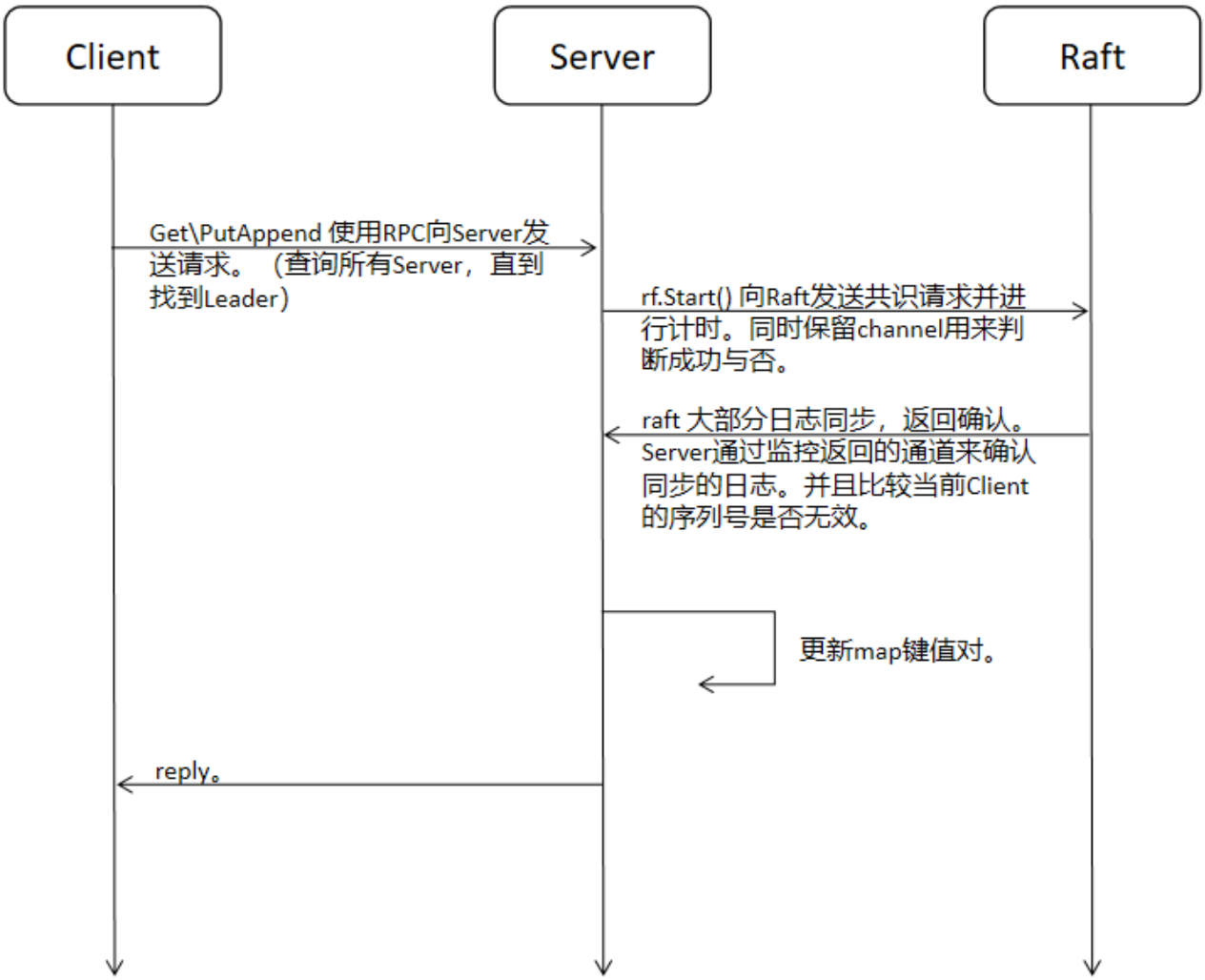
Lab3A

实验3的主要内容就是要利用Lab2中已经建立完成的raft机制来实现一个基本的KV存储的服务，能够实现三种操作，GET、PUT、APPEND。

我的理解是，这个服务类似Zookeeper，存在多个server，并且这多个server底层通过raft来达成共识。但是从要求来看，这些server内需要存储的内容是一样的，类似进行了一个多备份的场景。但是这三种操作，client都需要和被选举为leader的那个server进行交互。同时要求线性一致。按照要求，还需要跳过那些已经被处理的请求，所以应该和论文第八章所说的一样，需要有一个序列号来保证。



Client：

```
type Clerk struct {
    servers []*labrpc.ClientEnd

    mu         sync.Mutex
    leaderId   int
    clientId   int64
    sequenceId int64
    // You will have to modify this struct.
}
```

clerk就是client，其中要添加一些内容，leaderID为了验证交互的server是否是leader，sequenceId则是为了记录当前客户端已经发出的请求号，用来剔除由于网络不佳而后到的已经被处理的请求。

```
func (ck *Clerk) Get(key string) string {
    args := GetArgs{Key: key, ClientId: ck.clientId, SequenceId:
atomic.AddInt64(&ck.sequenceId, 1)}

    ck.mu.Lock()
    leaderId := ck.leaderId
    ck.mu.Unlock()

    for {
        reply := GetReply{}
        if ck.servers[leaderId].Call("KVServer.Get", &args, &reply) {
            if reply.Err == OK {
                return reply.Value
            } else if reply.Err == ErrNoKey {
                return ""
            }
        }
        //DPrintf("ID:%d not leader",ck.leaderId)
        ck.mu.Lock()
        ck.leaderId = (ck.leaderId + 1) % len(ck.servers)
        leaderId = ck.leaderId
        ck.mu.Unlock()
        time.Sleep(50* time.Millisecond)
    }
    // You will have to modify this function.
    return ""
}

//
// shared by Put and Append.
//
// you can send an RPC with code like this:
// ok := ck.servers[i].Call("KVServer.PutAppend", &args, &reply)
//
// the types of args and reply (including whether they are pointers)
// must match the declared types of the RPC handler function's
// arguments. and reply must be passed as a pointer.
//
func (ck *Clerk) PutAppend(key string, value string, op string) {
    // You will have to modify this function.
    args := PutAppendArgs{
        // You will have to modify this function.
        Key:        key,
        Value:      value,
        Op:         op,
        SequenceId: atomic.AddInt64(&ck.sequenceId, 1),
        ClientId:   ck.clientId,
    }
    //DPrintf("key:%v   value:%v  Op:%v sID:%d
cID:%d",args.Key,args.Value,args.Op,args.SequenceId,args.ClientId)
```

```
    ck.mu.Lock()
    leaderId := ck.leaderId
    ck.mu.Unlock()

    for {
        reply := PutAppendReply{}
        if ck.servers[leaderId].Call("KVServer.PutAppend", &args, &reply) {
            if reply.Err == OK {
                break
            }
        }
        ck.mu.Lock()
        ck.leaderId = (ck.leaderId + 1) % len(ck.servers)
        leaderId = ck.leaderId
        ck.mu.Unlock()
        time.Sleep(1 * time.Millisecond)
    }
}
```

GET操作和PUTAPPEND操作都比较简单，就是设置好需要通信的内容，包含操作类型，以及必不可少的序列号和客户端号，因为这两个号一起才能够唯一确定请求的顺序。

然后用一个循环，不断的去尝试和server通信，也就是通过RPC调用server的相关函数。因为所有请求只能通过leader进行实现，所以只有当server是leader时，并且leader回复了确认，才会退出循环，才说明这个请求已经成功了。

Server：

server比较复杂，还有一些关于channel的操作，以及需要阻塞的地方，对于Go语言并不太熟悉，所以会有些挣扎。

```
type KVServer struct {
    mu      sync.Mutex
    pmu      sync.Mutex

    me      int
    rf      *raft.Raft
    applyCh chan raft.ApplyMsg
    dead    int32 // set by Kill()

    maxraftstate int // snapshot if log grows this big

    // Your definitions here.
    sequenceMapper map[int64]int64
    requestMapper  map[int]chan Op
    kvStore        map[string]string
}
```

server也需要增加三个map，sequence用来记录对应client的已有序列号，request用来存放对应client的通道。channel本意可能更多是用来进行线程之间的数据交流。但是这里的作用其实更接近于暂存数据并方便触发server操作。

server的流程其实就是首先从client端接收到请求，然后将请求通过先前raft的Start函数再raft中达成共识，然后此时依据client号开设一个能够存放一个请求元数据的channel，同时设置定时器并使用select进行阻塞，等待事件发送。一个是channel中出现数据了，一个是超时了。

由于这里已经存在阻塞了，所以肯定不能在这里向channel写入数据。

所以要设置一个新的线程并运行一个死循环来不断监控。监控的方式其实很简单，就是监控自身底层raft机制向上层应用提供的apply，当出现这个apply时，就说明先前的请求已经可以提交，所以此时的server就可以真正执行这个操作，也就是修改kv存储。修改完之后，就可以按照clientid找到先前在阻塞的channel并向里面填充数据。

得到数据填充之后，先前的select激活，并判断此时的leader情况，通过之后就可以返回正确的reply给client。

client得到reply之后，就可以退出循环，结束这次的请求。

处理部分：

```
func (kv *KVServer) PutAppend(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    var isLeader bool

    clientOp := Op{OpType: args.Op, OpKey: args.Key, OpValue: args.Value, OpId:
args.SequenceId, ClientId: args.ClientId}
    kv.pmu.Lock()
    //DPrintf("Put append Op:%v key:%v value:%v seqId:%d clientID:%d"
,args.Op,args.Key,args.Value, args.SequenceId,args.ClientId)
    kv.pmu.Unlock()
    clientOp.Index, clientOp.Term, isLeader = kv.rf.Start(clientOp)

    if !isLeader {
        reply.Err = ErrWrongLeader
        return
    }


    // leader is found
    ch := kv.getChannel(clientOp.Index)

    defer func() {
        kv.mu.Lock()
        delete(kv.requestMapper, clientOp.Index)
        kv.mu.Unlock()
    }()

    timer := time.NewTicker(500 * time.Millisecond)
    defer timer.Stop()
    select {
    case op := <-ch:
        kv.mu.Lock()
        opTerm := op.Term
        kv.mu.Unlock()
        if clientOp.Term != opTerm {
            reply.Err = ErrWrongLeader
        } else {
            reply.Err = OK
        }
    case <-timer.C:
        reply.Err = ErrWrongLeader
    }
}
```

有了思路之后，代码就会比较轻松。

监控部分：

```
func (kv *KVServer) serverMonitor() {
    for {
        if kv.killed() {
            return
        }
        select {
        case msg := <-kv.applyCh:
            index := msg.CommandIndex
            term := msg.CommandTerm
            op := msg.Command.(Op)
            kv.mu.Lock()
            sequenceInMapper, hasSequence := kv.sequenceMapper[op.ClientId]
            op.Term = term
            kv.pmu.Lock()
            //DPrintf("~~~~~get command %v sID:%v  CID:%d   %d
",op.OpType,op.OpId,op.ClientId,sequenceInMapper)
            kv.pmu.Unlock()
            //qu chong
            if !hasSequence || op.OpId > sequenceInMapper {
                kv.pmu.Lock()
                //DPrintf("get command %v",op.OpType)
                kv.pmu.Unlock()
                switch op.OpType {
                case "Put":
                    kv.kvStore[op.OpKey] = op.OpValue
                case "Append":
                    kv.kvStore[op.OpKey] += op.OpValue
                }
                kv.sequenceMapper[op.ClientId] = op.OpId
                kv.pmu.Lock()
                //DPrintf("now key:%v value:%v",op.OpKey,kv.kvStore[op.OpKey])
                kv.pmu.Unlock()
            }
            kv.mu.Unlock()
            // send message to op chan
            kv.getChannel(index) <- op
        }
    }
}
```

```
lyj@ubuntu:~/Desktop/6.824/src/kvraft$ go test -run 3A
Test: one client (3A) ...
  ... Passed --  15.3  5  1680  319
Test: many clients (3A) ...
  ... Passed --  15.8  5  3684 1499
Test: unreliable net, many clients (3A) ...
  ... Passed --  18.6  5  3280  710
Test: concurrent append to same key, unreliable (3A) ...
  ... Passed --   1.4  3   160   52
Test: progress in majority (3A) ...
  ... Passed --   0.4  5    65    2
Test: no progress in minority (3A) ...
  ... Passed --   1.1  5   194    3
Test: completion after heal (3A) ...
  ... Passed --   1.1  5    96    3
Test: partitions, one client (3A) ...
  ... Passed --  22.9  5  2608  257
Test: partitions, many clients (3A) ...
  ... Passed --  23.8  5  4448 1413
Test: restarts, one client (3A) ...
  ... Passed --  19.8  5  2017  312
Test: restarts, many clients (3A) ...
  ... Passed --  20.4  5  4585 1487
Test: unreliable net, restarts, many clients (3A) ...
  ... Passed --  21.6  5  3435  763
Test: restarts, partitions, many clients (3A) ...
  ... Passed --  27.1  5  4552 1332
Test: unreliable net, restarts, partitions, many clients (3A) ...
  ... Passed --  27.0  5  3759  533
Test: unreliable net, restarts, partitions, many clients, linearizability checks (3A) ...
  ... Passed --  25.5  7  8396 1281
PASS
ok      _/home/lyj/Desktop/6.824/src/kvraft     242.294s
```

但是还是存在概率出错，并且错误全部集中在

partitions, 部分。

打印log时发现，似乎是在网络分区的情况下，leader选举出了问题。

多次尝试，确认是lab2中实现的raft机制存在bug，但是现在还没有找到具体原因。

所以后续的实验可能会先用别人已经完全正确的raft做替换，后续再去找自己做的raft的问题