**中国科学技术大学软件学院**

**软件工程实验项目环节**

**结题报告**

**项 目 名 称：基于multi-raft的分布式数据库的设计与实现**

**成 员 名 单：吴桐华、戴明成、王奇**

**导 师： 张曙**

**工 程 领 域： 数据库**

**研 究 方 向： 分布式数据库**

**结 题 时 间： 2022-06-27**

**中国科学技术大学软件学院**

**填表日期：2022年6月27日**

# 1 项目概述

## 项目名称

基于Multi-Raft的分布式数据库。

## 项目背景

自数据库的诞生以来，随着系统访问量的不断上升，单体数据库的读写性能已经不足以支撑日常访问。因此分库分表中间件开始火热起来，借此缓解单数据库的性能问题。但是分库分表中间件不支持事务，要保证数据一致性又要借助于分布式事务中间件。因此业界继续一种高效的解决访问，于是分布式数据库登上历史舞台。

分布式数据库是数据库技术与网络技术相结合的产物，在1979年，世界上第一个分布式数据库SDD-1诞生于美国CCA公司。分布式数据库真正的大爆发是从2005年开始，Google发表了三篇经典分布式论文MapReduce、BigTable、GFS。其中BigTable 是 Google 内部使用的分布式数据库，构建在 GFS 的基础上，弥补了分布式文件系统对于小对象的插入、更新、随机读请求的缺陷。而Hbase则是基于HDFS的针对BigTable的开源实现版本，它本身并不存储数据，而是通过HDFS。另外，Hbase和BigTable都不支持跨行事务，Google又在其上开发了Percolator这样的事务层。

分布式数据库的又一次大跨越是在2012-2013年，Google相继发布了Spanner和F1两篇论文，融合了关系模式和NoSQL,Spanner和F1的出现标志着NewSQL首次在生产环境中提供服务。并且提供了：SQL支持、ACID事务、水平扩展、AutoFailover、多机房异地容灾。基于Paxos带来的强大能力，其写入延迟大约在100ms左右。Spanner的大获成功引来了一大批的追随者，例如：CockroachDB 和TiDB。

## 项目简介

本软件系统是一个可横向扩展的、高可用的key-value存储系统。在KV存储方面，使用HashMap的数据结构。未来计划使用LSM的存储结构进行存储，LSM树是一种广泛应用的一种写效率极高的存储结构。

在分布式一致性方面使用Raft协议同时Raft协议与上述的自行封装的K-V存储接口相关联。与此同时，本项目还计划通过并行复制日志、预投票pre-vote、Hibernate Region等方式与业界最先进的Raft协议实现、优化方式靠拢。

本项目采用改进扩展Raft协议为Raft的改进版本：multi-Raft协议，以克服Raft协议本身设计导致的仅有一个Leader带来的性能瓶颈、提升系统并发度。

# 开发过程

## 项目计划

软件系统的开发计划如下：

|  |  |
| --- | --- |
| 时间 | 内容 |
| 2021.11.1-2021.11.25 | 查阅相关文献资料，进行软件开发方案设计，确定初步的开发计划。 |
| 2021.11.25-2021.12.25 | 完成第一阶段的开发，具体任务为：实现分布式数据库的分布式处理部分，即实现Raft一致性算法，其中包括Raft选举、Raft日志备份、Raft与上层key-value存储系统交互接口、Raft Configuration Change。 |
| 2021.12.26-2022.1.25 | 实现基于Raft一致性协议的Key-value可容错数据库。 |
| 2022.1.26-2022-2.25 | 完成第二阶段的开发，具体任务为：实现multi-Raft，包括分片及数据迁移等。 |
| 2022.2.26-2022-3.25 | 总结与测试，重构部分代码。修改测试发现的bug,最终在能够模拟复杂网络环境的测试脚本下连续通过100次以上。 |

## 开发环境

1. 处理器型号：Intel 64位处理器；
2. 内存容量：不小于50G；
3. 操作系统：ubuntu20.04.3（或linux其它版本系统）；
4. 编程语言：Golang 1.15；
5. 版本控制器：Git。

## 开发步骤

1. 完成需求分析，详见需求分析文档；
2. 完成概要设计，详见概要设计文档；
3. 完成详细设计，详见详细设计文档；
4. 完成编码，具体见代码工程文件；
5. 测试，详见技术报告。

## 人员分工

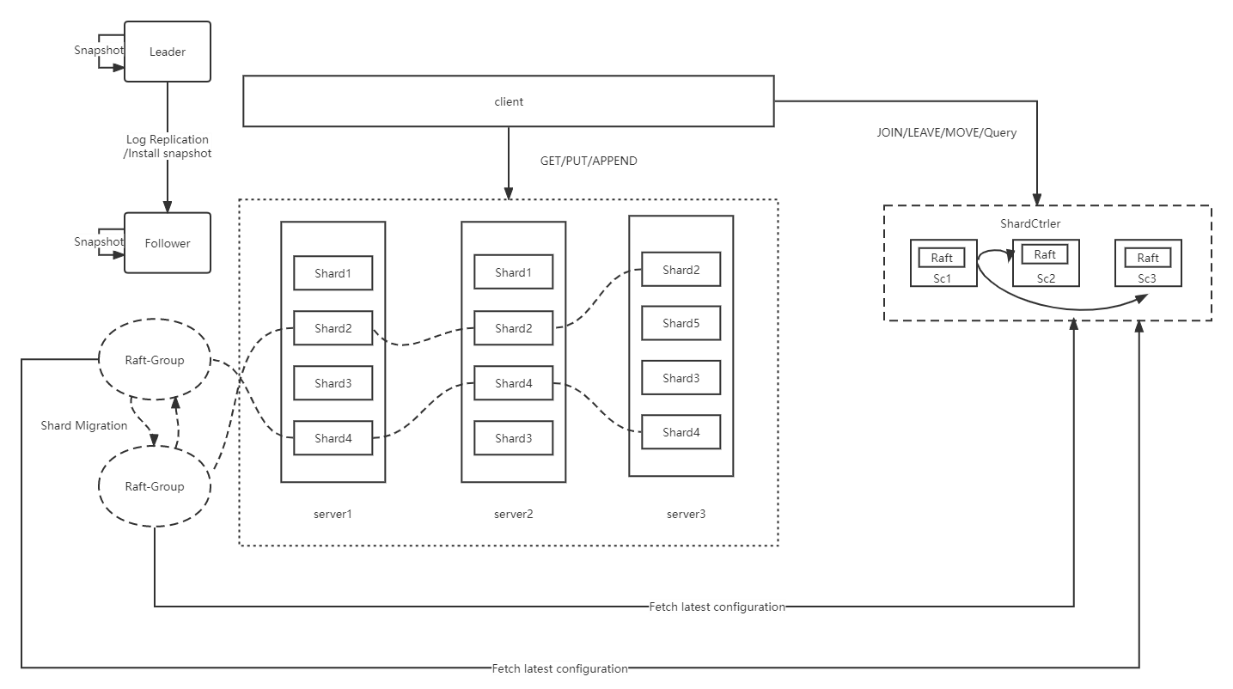
|  |  |  |
| --- | --- | --- |
| 姓名 | 学号 | 项目中的分工 |
| 吴桐华 | SA21225459 | Raft协议实现 |
| 戴明成 | SA21225150 | Multi-raft与存储层接口实现 |
| 王奇 | SA21225429 | 分布式事务功能实现 |

## 最终成果

实现一个功能完备的分布式key-value存储系统，支持Put/Append/Get/ 三种操作（未来还可以拓展多种其他操作，例如Remove、Scan等），能够通过脚本测试网络的提交测试，成功通过连续数百个测试实例的验证。

# 项目实现

## 项目架构



项目架构图如图1所示，本项目采用三副本架构。数据被划分为了多个分片，存储相同数据的分片组成一个Raft组，其独立运行Raft协议。每个Raft组会定时向ShardCtrler拉取最新的配置信息，若发现配置变更，需要向其他Raft组发送数据迁移的请求。模块划分图中的ShardCtrler用于管理集群的配置（包括分片属于的组号，服务器列表等），其也是三副本架构，运行Raft协议，客户端会向ShardCtrler发送RPC请求，Leader会将请求对应的操作写入日志条目中，并向从节点发送追加日志的RPC请求。

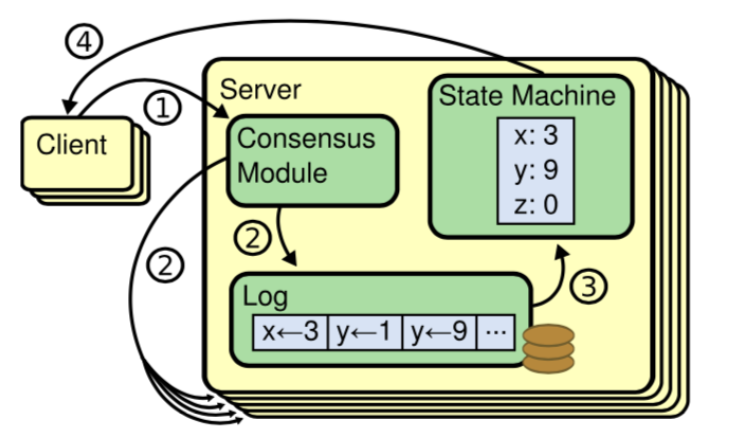
## 具体实现

### Raft算法

共识算法允许一组节点像一个整体一样一起工作，即使其中一些节点出现故障也能够继续工作下去，其正确性主要源于复制状态机的性质：任何初始状态一样的状态机，如果执行的命令序列一样，则最终达成的状态也一定一致。所以要保障状态机最终能保持一致，就是要让状态机初始状态完全一致和执行命令序列完全一致。

Paxos算法就是一个划时代的共识算法。在Raft出现之前的10年里，Paxos几乎统治者共识算法这一领域：因为绝大多数共识算法的实现都是基于Paxos或受其影响。但是Paxos算法十分复杂，难以理解，并且其自身的算法结构还需要进行大幅的修改才能应用到实际的系统中。由于Paxos的描述和现实差距太大，最终人们只能实现一套未经证实的类Paxos算法。

基于以上背景，Diego Ongaro在就读博士期间，深入研究Paxos协议后提出了Raft协议，旨在提供更为易于理解的共识算法。

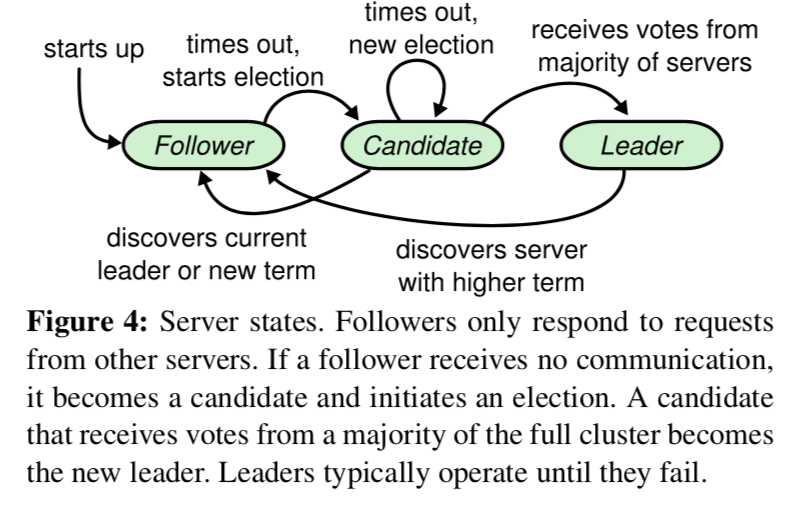


#### 选举

Raft将所有节点分为了三个身份，Leader, candidate及follower。集群内只会有一个Leader，其负责发起心跳，响应客户端、创建日志条目，同步日志条目等。Candidate是选举过程中的临时角色，由follower转化而来，发起投票参与竞选。而Follower接收Leader的心跳和日志条目同步数据，可能会投票给candidate。每个follower都会维护一个随机的定时器，当定时器超时就会发起选举，而在定时器未超时期间，若收到了leader的心跳或是发现有节点的term比自己更大，则会重置计时器，继续保持follower的状态。

本项目定义了三个常量来表示Raft的三种状态。

const *FOLLOWER*, *CANDIDATE*, *LEADER* = 0, 1, 2



#### 日志同步

Leader被选举后，则负责所有客户端的请求，每个客户端请求都包含一个命令，该命令可作用到状态机。Leader收到客户端请求后，会生成一个日志条目，其包含index、term、cmd，将其追加到自己的日志数组末尾后，会向所有的节点广播该日志条目。若follower同意接收，则在将日志条目添加到自己的日志后，返回同意。如果大部分节点都成功接收，则该日志条目将被Leader提交，然后提交的信息也会被Leader发送给其他节点。

#### 日志压缩

随着系统的运行，Raft日志条目不断增长，其会占用更多的空间，并且需要花费更多的时间进行重放。所以需要一种机制去压缩日志条目，而这就是快照技术。

快照会包含如下信息:

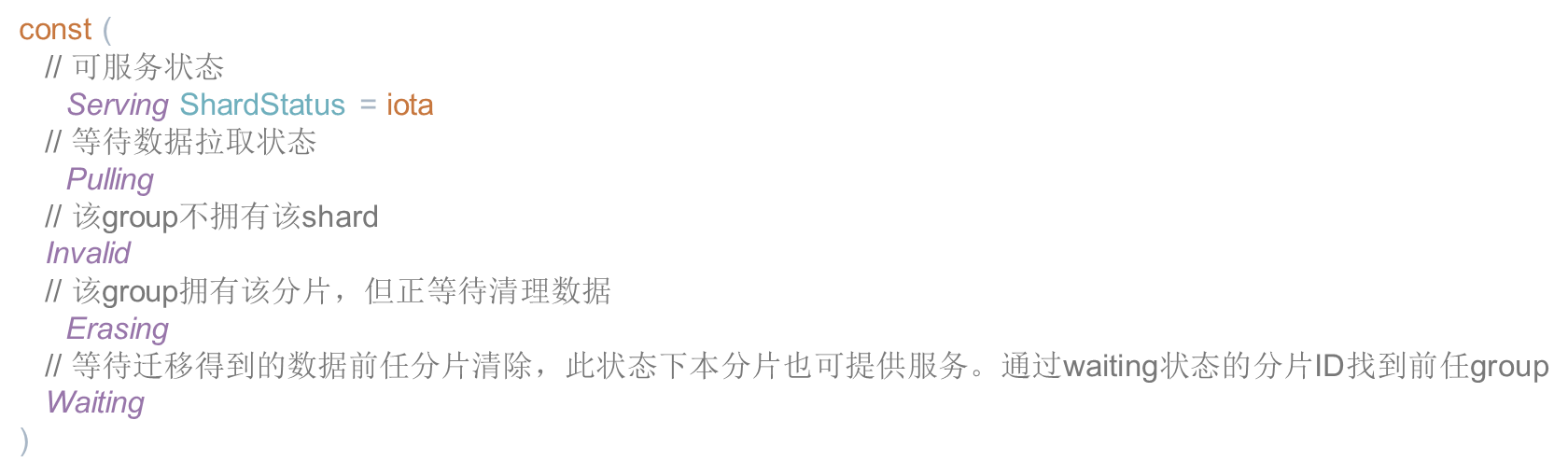
1. 状态机当前状态
2. 状态机最后一条应用的日志条目对应的index和term。
3. 集群的配置信息。
4. 去重表。

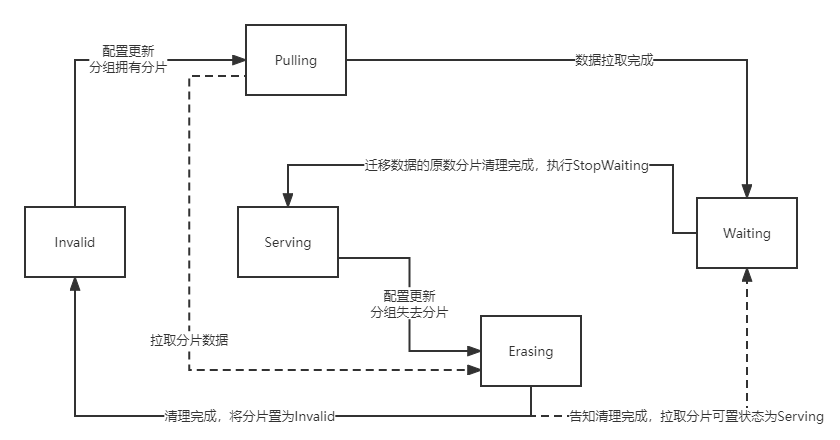
在本项目中，每个节点独立地当日志条目达到一定阈值时进行快照。随着系统的运行可能出现一种情况：leader的日志条目达到阈值，于是对已应用的日志条目进行了压缩，但此时某个节点甚至还没有得到最初的那几个日志条目。Leader无法追加快照之前的日志条目，于是只能通过发送快照的方式进行“日志同步”。

### Multi-raft的设计与实现

#### 分片

为了使问题更加清晰，本项目定义了分片的几种状态，分片的几种状态可能在某些时刻进行相互转化。





初始时，Raft组不拥有任何分片，所以所有分片的状态均为Invalid。随后，Raft组拉取最新配置，发现自己拥有了某些分片，于是从其他Raft组的分片中拉取数据，分片的状态变为Pulling，被拉取的分片状态置为Erasing。拉取完成后，拉取分片状态由Pulling变更为Waiting状态。被拉取分片清理完成后会告知拉取分片清理完成，拉取分片将由Waiting状态转为正常Serving状态。分片状态为Serving或者Waiting状态时可正常对外提供服务。而其他状态的分片不能提供服务。

在本项目中，采用哈希的方式将不同key的键值对散列到不同的分片上。当客户端发起Get或者PUT/APPEND请求时，服务端通过同样的哈希函数进行相应的操作。

type Session struct {  
 LastRequestedId int64  
 Err Err  
 //Result NotifyMsg  
}  
type Shard struct {  
 KV map[string]string  
 Status ShardStatus  
 // LastRequestedId map[int64]int64  
 LastSession map[int64]\*Session  
}

#### 日志类型

为了使问题更加清晰，本项目定义了如下日志类型。

const (  
 *Operation* CommandType = iota  
 *ConfigChange  
 Insert  
 Erase  
 StopWaiting  
 Empty*)

Operation表示是对数据库的操作命令，ConfigChange表示配置更新的日志条目。Insert表示拉取到数据，需要将其复制到本分片上。Erase表示需要对分片的数据进行清除。StopWaiting表示分片清理完成，拉取分片可由Pulling状态转为Serving状态。Empty表示空日志，不进行任何操作，其目的是为了让集群快速达到可用的状态。

#### 配置管理器

ShardCtrler就是一个高可用的集群配置管理器，其主要记录raft组对应的服务器以及每个分片属于哪一个raft组。ShardCtrler可进行四种操作：(1) Join，加入分组。(2)Leave，删除分组。(3) Move，将分片从一个分组移动到另一个分组。(4) Query，查询最新的配置。

配置维护三个属性，一是配置的num，代表版本号，然后有分片到group id的映射，还维护了group id到服务器数组的映射。

type Config struct {  
 Num int // config number  
 Shards [*NShards*]int // shard -> gid  
 Groups map[int][]string // gid -> servers[]  
}

不仅仅是客户端需要查询最新的配置，每个Raft组也会周期性地检查是否有更新的配置（本项目中50ms尝试拉取一次）。

#### 分片迁移

本项目采取“拉取”的方式完成迁移。本项目单独建立了一个协程完成分片拉取工作，其周期性地找到状态为Pulling的分片，并通过上一次config找到分片的上一任raft组id和要拉取的分片，然后可以并行地进行拉取数据。

// 向另外的节点拉取shard  
func (kv \*ShardKV) tryPullShard() {  
 kv.lock("tryPullShard")  
 oldGidToShards := kv.getOldGidToShards(*Pulling*)  
 kv.log("pulling groupToshards: %+v", oldGidToShards)  
 var wait sync.WaitGroup  
 for gid, shardIds := range oldGidToShards {  
 wait.Add(1)  
 // 前任config的servers  
 kv.log("in trypPullShard, gid: %d, shardId: %v", gid, shardIds)  
 kv.log("prevConfig: %v", kv.prevConfig)  
 go func(configNum int, servers []string, shardIds []int) {  
 args := PullShardArgs{  
 ShardIds: shardIds,  
 ConfigNum: configNum,  
 }  
 kv.log("loop servers to send pull shards, servers: %v", servers)  
 for \_, server := range servers {  
 reply := PullShardReply{}  
 srv := kv.make\_end(server)  
 if srv.Call("ShardKV.PullShard", &args, &reply) && reply.Err == *OK* {  
 kv.log("start insert command, %v", reply)  
 kv.rf.Start(newRaftLogCommand(*Insert*, reply))  
 break  
 }  
 }  
 wait.Done()  
 }(kv.currentConfig.Num, kv.prevConfig.Groups[gid], shardIds)  
  
 }  
 kv.unlock("tryPullShard")  
 wait.Wait()  
}

#### 分片清理

分片清理也需要一个单独的协程。该协程定时找出处于Waiting状态的分片，通过lastConfig找到对应的应该清理的分片，然后并行地清除数据。

// 先新config拉取，后通过旧config erase  
func (kv \*ShardKV) tryEraseShard() {  
 kv.lock()  
 // 找到gid->shardId对应关系  
 oldGidToShards := kv.getOldGidToShards(*Waiting*)  
 kv.log("erasing groupToShards: %v", oldGidToShards)  
 currentConfigNum := kv.currentConfig.Num  
 wg := sync.WaitGroup{}  
 for gid, shards := range oldGidToShards {  
 wg.Add(1)  
 servers := kv.prevConfig.Groups[gid]  
 go func(configNum int, servers []string, shardsIds []int) {  
 defer wg.Done()  
 for \_, server := range servers {  
 srv := kv.make\_end(server)  
 args := EraseShardArgs{  
 ConfigNum: configNum,  
 ShardIds: shardsIds,  
 }  
 reply := EraseShardReply{}  
 if srv.Call("ShardKV.EraseShard", &args, &reply) && reply.Err == *OK* {  
 kv.rf.Start(newRaftLogCommand(*StopWaiting*, args))  
 break  
 }  
 }  
 }(currentConfigNum, servers, shards)  
 }  
 kv.unlock()  
 wg.Wait()  
}

分片数据清理完成后，向拥有waiting状态分片的raft组发送StopWaiting命令，让其更改分片状态为Serving。

#### 读写操作

在进行读写操作之前，服务器会先将请求的key散列到某个分片上。当当前raft组在当前config下负责管理分片，该分片的状态为Serving或Waiting时，本raft组可以为该分片提供正常的读写服务，否则会返回ErrWrongGroup使客户端重新发送请求。

为了保证线性一致性，对每个分片维护了一个Session字段，其保存一个客户端ID到最后一次请求ID的映射及命令操作结果。执行操作前，服务端为先进行判断是否为重复操作，若为重复操作则直接返回最后一次请求的结果，否则立即执行相应操作，并更新Session字段。

func (kv \*ShardKV) shardMatches(shardId int) bool {  
 kv.log("shardTogid:%v, gid:%v, status: %v", kv.currentConfig.Shards[shardId], kv.gid, kv.stateMachines[shardId].Status)  
 return kv.currentConfig.Shards[shardId] == kv.gid &&  
 (kv.stateMachines[shardId].Status == *Serving* || kv.stateMachines[shardId].Status == *Waiting*)  
}  
func (kv \*ShardKV) requestIsDuplicate(clientId int64, requestId int64, shardId int) bool {  
  
 lastSession, ok := kv.stateMachines[shardId].LastSession[clientId]  
 if ok {  
 kv.log("requestIsDuplicate func: lastRequestId: %d requestId: %d", lastSession.LastRequestedId, requestId)  
 } else {  
 kv.log("requestIsDuplicate func: lastRequestId: null, requestId: %d", requestId)  
 }  
 if !ok || requestId > lastSession.LastRequestedId {  
 return false  
 }  
 return true  
}

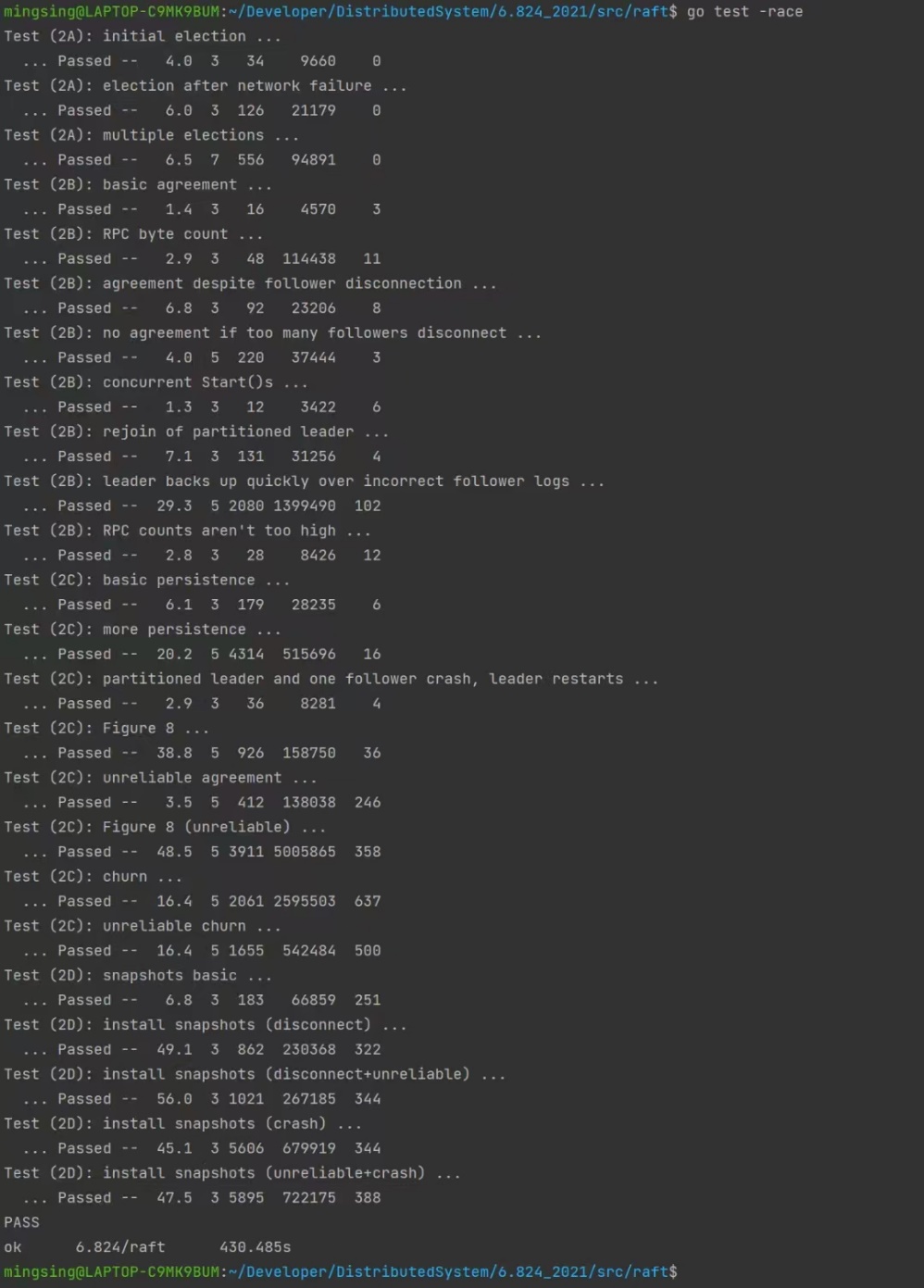
#### 整体结构

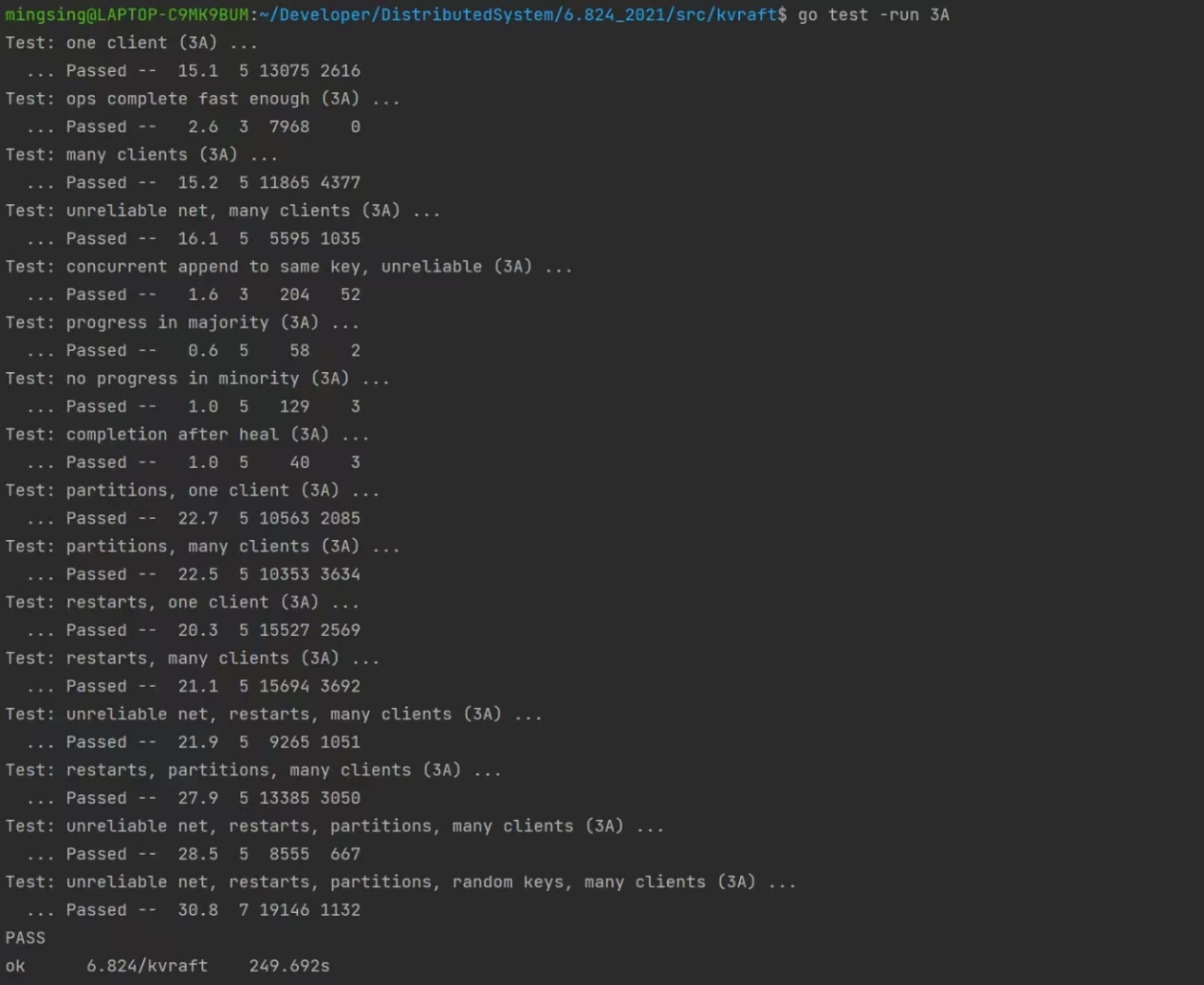
整体流程如代码所示。cmdApply函数不断从状态机通道中读取命令，并从命令结构中读取命令类型信息，并执行对应的操作。当发现日志条目达到一定阈值时，会进行快照处理。

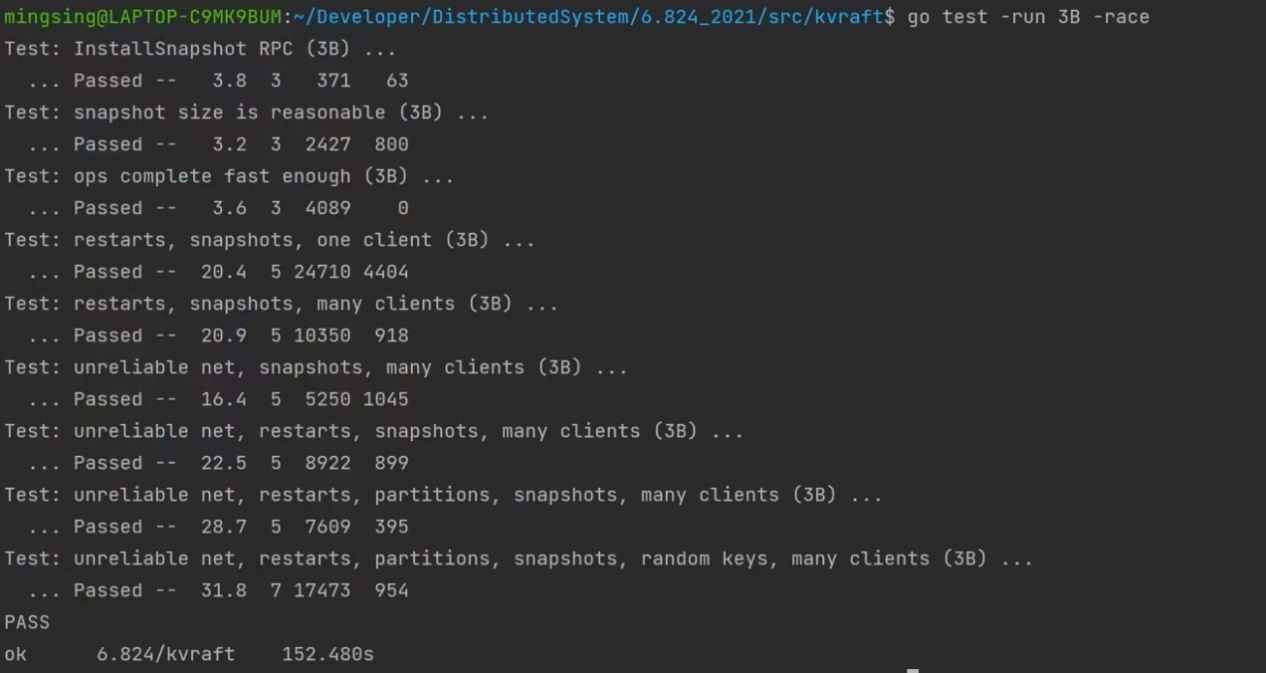
func (kv\* ShardKV) cmdApply() {  
 for !kv.killed(){  
 select {  
 case msg := <-kv.applyCh:  
  
 if msg.CommandValid {  
 kv.log("get commandMsg ready to apply: %v, msgIndex: %d", msg, msg.CommandIndex)  
 command := msg.Command.(RaftLogCommand)  
 kv.lock("cmdApply")  
 res := NotifyMsg{}  
 switch command.CommandType {  
 case *Operation*:  
 op := command.Data.(Op)  
 res = kv.applyOperation(op)  
 if notify, okk := kv.mapNotify[msg.CommandIndex]; okk {  
 notify <- res  
 }  
 //kv.log("operation done, res: %v", res)  
 case *ConfigChange*:  
 latestConfig := command.Data.(shardctrler.Config)  
 kv.applyConfigChange(&latestConfig)  
 //kv.log("apply successfully, currentConfig: %v", kv.currentConfig)  
 case *Insert*:  
 pullShardReply := command.Data.(PullShardReply)  
 kv.applyInsertShards(&pullShardReply)  
 //kv.log("insert done")  
 case *StopWaiting*:  
 eraseShardArgs := command.Data.(EraseShardArgs)  
 kv.applyStopWaiting(&eraseShardArgs)  
 //kv.log("stopwaiting done")  
 case *Erase*:  
 eraseShardArgs := command.Data.(EraseShardArgs)  
 res = kv.applyEraseShard(&eraseShardArgs)  
 //kv.log("erase done, res: %v", res)  
 if notify, okk := kv.mapNotify[msg.CommandIndex]; okk {  
 notify <- res  
 }  
 case *Empty*:  
  
 }  
 kv.checkSnapshot(msg.CommandIndex)  
 kv.unlock("cmdApply")  
 } else {  
 if msg.SnapshotValid {  
 kv.lock("condInstall")  
 if kv.rf.CondInstallSnapshot(msg.SnapshotTerm, msg.SnapshotIndex, msg.Snapshot) {  
 kv.getPersistSnapshot()  
 }  
 kv.unlock("condInstall")  
 }  
 }  
 }  
 }  
}

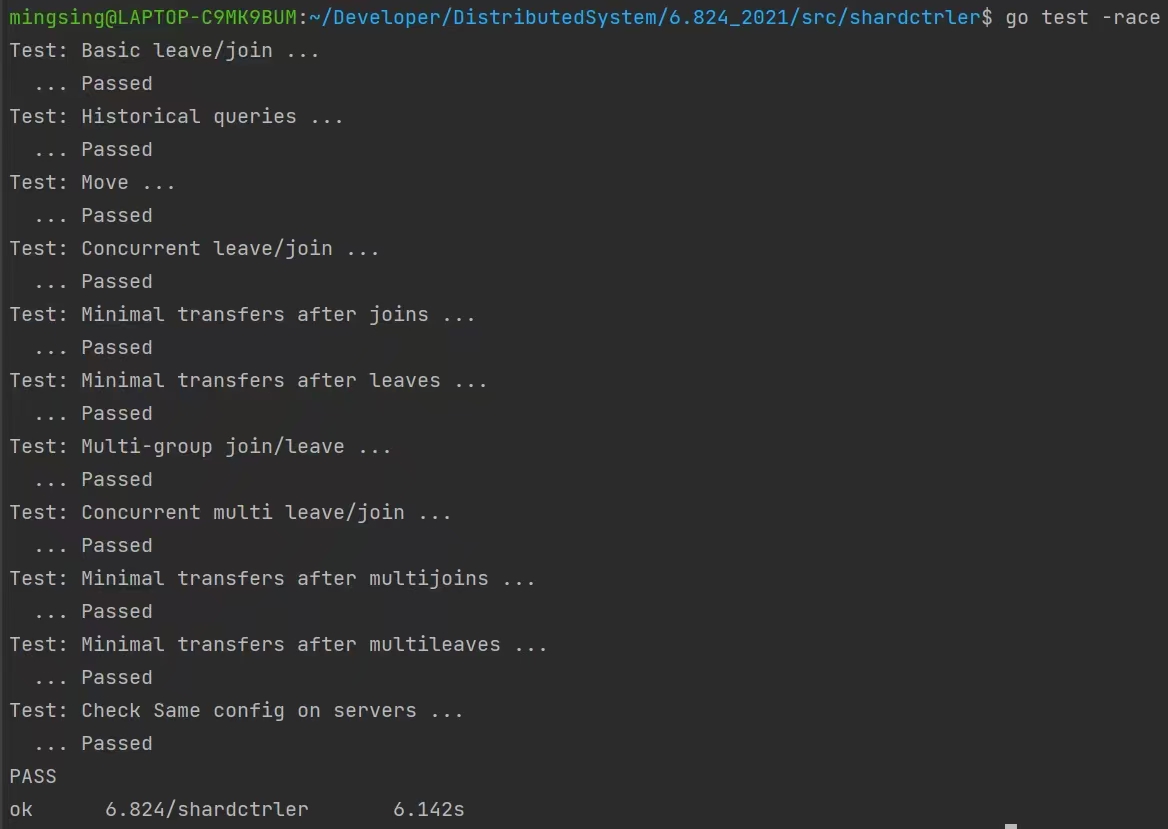
## 项目结果与测试

经过调试，使用测试脚本模拟复制网络环境（包括并发环境、网络分区环境、节点宕机及恢复等），最终能够连续通过数百次各种环境的测试。测试结果截图（测试脚本见源文件）：









# 项目总结

## 不足之处

项目中的分布式事务部分，由于知识的深度过高和时间不充足等问题，我们尚未全部实现。整体数据库项目中仅实现了基础性的操作功能，无法直接放入到实际的生产环境使用，后期仍然需要进行做一些版本迭代和功能上的完善。

## 心得体会

在这次工程实践过程中，我们学到了很多东西，第一次深刻的体会到了什么叫做用工程化的思想来编写数据库软件，以前自己也写过一些小型软件，没有做过大型的项目，直到这次工程实践我担任组长并组织组员共同完成“分布式数据库”这个项目，本次与别人合作过程中，发现运用工程化的思想来做是如此的有必要。

从这里，我才真正的意识到实施一个软件工程并不是说简单的会编码就能够解决问题的，我们更多的精力不是放在编码上，编码只是一个很小的模块，只占到那么小的一个部分。这个事实在很大程度上颠覆了我以前的思想，在我以前的认识中，似乎整个软件就是编码，深度实践的过程中，体会到需要使用软件工程的思想来完成这个工作。

真正开始工作之前，我们费了很多的时间来完成一些前期调研工作，如需求分析和可行性分析，这块工作在别人看来可能是相对无关紧要，甚至是多于的。可是，我现在算是深深地明白了磨刀不误砍柴工的道理，这些工作的完成太有必要了，太重要了，要想你的软件有用有市场，能被别人接受和认可，在进行过程中不会出现崩溃性的问题，这些工作缺一不可。

在本次工程实践中，我们最大的收获是从0到1的能力，面对任何未知复杂的问题，我们都能够逐步去解决它，这个过程就是我们逐步提高的过程。