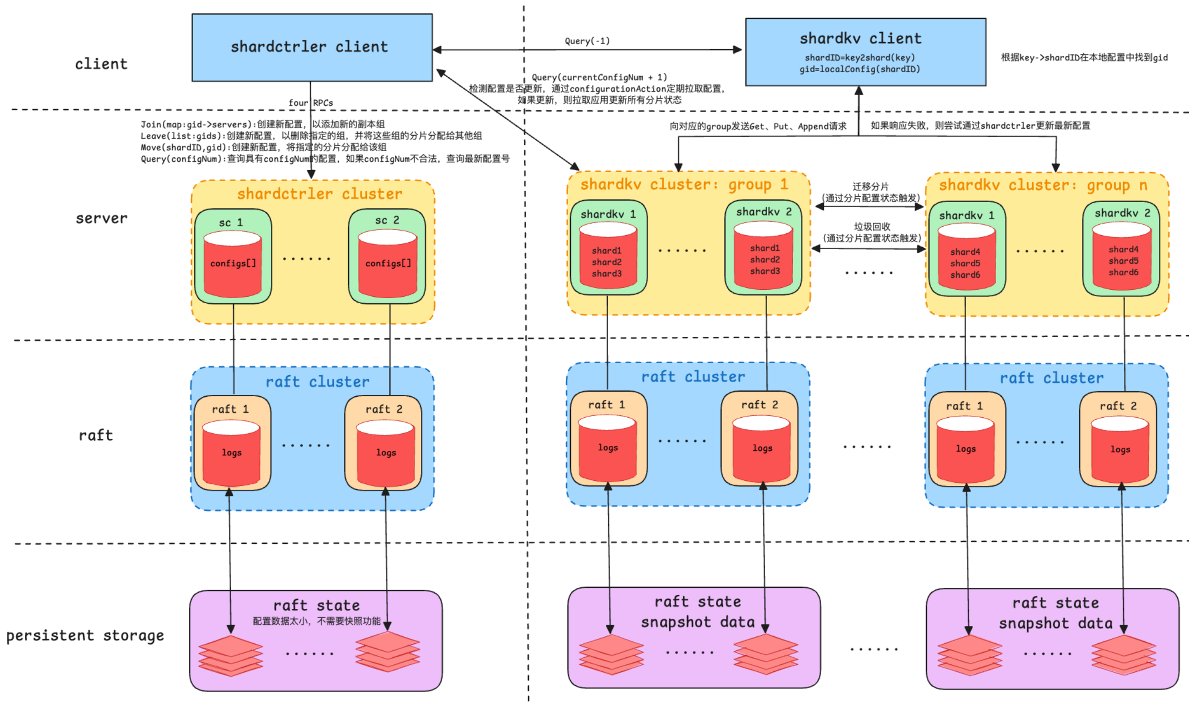
Lab5A

前言

本实验要求构建一个键 / 值存储系统，该系统能够将键 “分片” 或分区到一组副本组上。分片是键 / 值对的子集，例如所有以“a”开头的键可能是一个分片等，通过分片可提高系统性能，因为每个副本组仅处理几个分片的放置和获取，并且这些组并行操作。

系统有两个主要组件：一组副本组和分片控制器。每个副本组使用 Raft 复制负责部分分片的操作，分片控制器决定每个分片应由哪个副本组服务，其配置会随时间变化。客户端和副本组都需咨询分片控制器来找到对应关系。系统必须能在副本组间转移分片，以平衡负载或应对副本组的加入和离开。

主要挑战在于处理重新配置，即分片到组的分配变化，且要确保任何时候每个分片只有一个副本组在处理请求，同时重新配置还需要副本组间的交互（分片移动）。本实验只允许通过 RPC 进行客户端和服务器间的交互。实验架构与许多其他系统类似，但相对简单。实验需使用相同的 Raft 实现，完成后需通过相关测试。



shardctrler

Common

该部分主要包含一些命令参数，其中包括：

1. Config：用于保存shardcrler 的配置信息
2. OpType：用于表示客户端的命令请求
3. Err：用于shardcrler 服务器回复客户端命令的执行情况
4. CommandReply：服务端命令回复RPC参数
5. CommandArgs：客户端命令发送RPC参数
6. OperationContext：用于保存每个客户端最近一次执行完操作的命令ID和回复

```
//src/shardctrler/common.go
package shardctrler
```

```go
import (
	"fmt"
	"time"
)

//
// Shard controller: assigns shards to replication groups.
//
// RPC interface:
// Join(servers) -- add a set of groups (gid -> server-list mapping).
// Leave(gids) -- delete a set of groups.
// Move(shard, gid) -- hand off one shard from current owner to gid.
// Query(num) -> fetch Config # num, or latest config if num==-1.
//
// A Config (configuration) describes a set of replica groups, and the
// replica group responsible for each shard. Configs are numbered. Config
// #0 is the initial configuration, with no groups and all shards
// assigned to group 0 (the invalid group).
//
// You will need to add fields to the RPC argument structs.
//

// The number of shards.
const NShards = 10

const ExecuteTimeout = 500 * time.Millisecond

// A configuration -- an assignment of shards to groups.
// Please don't change this.
type Config struct {
	Num    int              // config number 初始配置的编号为 0，每次通过 Join、Leave 或 Move 操作创建新配置时，编号加 1
	Shards [NShards]int     // shard -> gid  表示分片 i（从 0 到 NShards-1）当前由哪个副本组（通过 GID 标识）负责。
	Groups map[int][]string // gid -> servers[] 记录系统中活跃的副本组及其服务器地址列表。
	//值（[]string） 该副本组中所有服务器的地址列表（例如 ["server1", "server2"]）。
}

func DefaultConfig() Config {
	return Config{
		Groups: make(map[int][]string),
	}
}

func (cf Config) String() string {
	return fmt.Sprintf("{Num:%v,Shards:%v,Groups:%v}", cf.Num, cf.Shards, cf.Groups)
}

type OpType uint8

const (
	Join OpType = iota
	Leave
	Move
	Query
)
```

```go
func (op OpType) String() string {
	switch op {
	case Join:
		return "Join"
	case Leave:
		return "Leave"
	case Move:
		return "Move"
	case Query:
		return "Query"
	default:
		panic(fmt.Sprintf("unknown operation type %d", op))
	}
}

type Err uint8

const (
	OK Err = iota
	ErrWrongLeader
	ErrTimeout
)

func (e Err) String() string {
	switch e {
	case OK:
		return "OK"
	case ErrWrongLeader:
		return "ErrWrongLeader"
	case ErrTimeout:
		return "ErrTimeout"
	default:
		panic(fmt.Sprintf("unknown error type %d", e))
	}
}

type CommandReply struct {
	Err    Err
	Config Config
}

func (reply CommandReply) String() string {
	return fmt.Sprintf("{Err:%v,Config:%v}", reply.Err, reply.Config)
}

type CommandArgs struct {
	//
	Servers map[int][]string // for Join 用于将一组服务器加入到分布式系统中某个组或集群中。
	GIDs    []int            //for Leave  用于从分布式系统中移除某些组或服务器
	Shard   int              //for Move Shard 可能代表一个特定的分片 ID（shard ID)
	//Shards=[1,1,1,2,2,2,3,3,3,3]:
	// 表示 10 个分片分配给 3 个副本组:
	// 分片 0-2: GID 1。
	// 分片 3-5: GID 2。
	// 分片 6-9: GID 3。
```

```
    GID       int //for Move 用于在分布式系统中移动数据或责任（如将某个组的数据迁移到另一个组）
    Num       int //for Query 用于查询分布式系统的状态或信息
    Op        OpType
    ClientId  int64
    CommandId int64
}

func (args CommandArgs) String() string {
    switch args.Op {
    case Join:
        return fmt.Sprintf("{Servers:%v,Op:%v,ClientId:%v,CommandId:%v}", args.Servers, args.Op, args.ClientId, args.CommandId)
    case Leave:
        return fmt.Sprintf("{GIDs:%v,Op:%v,ClientId:%v,CommandId:%v}", args.GIDs, args.Op, args.ClientId, args.CommandId)
    case Move:
        return fmt.Sprintf("{Shard:%v,GID:%v,Op:%v,ClientId:%v,CommandId:%v}", args.Shard, args.GID, args.Op, args.ClientId, args.CommandId)
    case Query:
        return fmt.Sprintf("{Num:%v,Op:%v,ClientId:%v,CommandId:%v}", args.Num, args.Op, args.ClientId, args.CommandId)
    default:
        panic(fmt.Sprintf("unknown operation type %d", args.Op))
    }
}

type OperationContext struct {
    MaxAppliedCommandId int64
    LastReply           *CommandReply
}
type Command struct {
    *CommandArgs
}
```

Client

主要用于组织用户传来的操作和参数，调用Server端的RPC执行命令

```
//src/shardctrler/client.go
package shardctrler

//
// Shardctrler clerk.
//

import (
    "crypto/rand"
    "math/big"

    "6.5840/labrpc"
)

type Clerk struct {
    servers []*labrpc.ClientEnd
    // Your data here.
    leaderId  int64
```

```go
    clientId  int64
    commandId int64
}

func nrand() int64 {
    max := big.NewInt(int64(1) << 62)
    bigx, _ := rand.Int(rand.Reader, max)
    x := bigx.Int64()
    return x
}

func MakeClerk(servers []*labrpc.ClientEnd) *Clerk {
    return &Clerk{
        servers:   servers,
        leaderId:  0,
        clientId:  nrand(),
        commandId: 0,
    }
}

func (ck *Clerk) Query(num int) Config {
    args := &CommandArgs{Op: Query, Num: num}
    return ck.Command(args)
}

func (ck *Clerk) Join(servers map[int][]string) {
    args := &CommandArgs{Op: Join, Servers: servers}
    ck.Command(args)
}

func (ck *Clerk) Leave(gids []int) {
    args := &CommandArgs{Op: Leave, GIDs: gids}
    ck.Command(args)
}

func (ck *Clerk) Move(shard int, gid int) {
    args := &CommandArgs{Op: Move, Shard: shard, GID: gid}
    ck.Command(args)
}

func (ck *Clerk) Command(args *CommandArgs) Config {
    args.ClientId, args.CommandId = ck.clientId, ck.commandId
    for {
        var reply CommandReply
        if !ck.servers[ck.leaderId].Call("ShardCtrler.Command", args, &reply) ||
reply.Err == ErrWrongLeader || reply.Err == ErrTimeout {
            ck.leaderId = (ck.leaderId + 1) % int64(len(ck.servers))
        } else {
            DPrintf("命令%v执行成功", args.CommandId)
            ck.commandId++
            return reply.Config
        }
    }
}
```

Server

ShardCtrler服务器结构体

```
type ShardCtrler struct {
	mu      sync.RWMutex       //确保并发安全，保护共享状态。
	me      int                //标识服务器，辅助 Raft 或调试
	rf      *raft.Raft         //  Raft 实例，管理一致性协议
	applyCh chan raft.ApplyMsg //Raft 日志提交通道，连接状态机

	// Your data here.
	dead           int32                         //控制服务器终止，优雅退出
	stateMachine   ConfigStateMachine            //核心状态机，管理配置历史和操作
	LastOperations map[int64]OperationContext //去重客户端请求，防止重复执行。
	notifyChans    map[int]chan *CommandReply //异步通知 Raft 提交结果，提升并发性
}
```

初始化并开启服务器

```
func StartServer(servers []*labrpc.ClientEnd, me int, persister *raft.Persister)
*ShardCtrler {

	// Your code here.
	InitLogger()
	labgob.Register(Command{})
	apply := make(chan raft.ApplyMsg)

	sc := &ShardCtrler{
		rf:             raft.Make(servers, me, persister, apply),
		applyCh:        apply,
		stateMachine:   NewMemoryConfigStateMachine(),
		LastOperations: make(map[int64]OperationContext),
		notifyChans:    make(map[int]chan *CommandReply),
		dead:           0,
	}
	go sc.applier()
	return sc
}
```

处理客户端的远程RPC调用

```
func (sc *ShardCtrler) Command(args *CommandArgs, reply *CommandReply) {
	sc.mu.RLock()
	//检查命令是否为重复请求（仅适用于非查询命令）
	if args.Op != Query && sc.isDuplicateRequest(args.ClientId, args.CommandId)
{
		lastReply := sc.LastOperations[args.ClientId].LastReply
		reply.Config, reply.Err = lastReply.Config, lastReply.Err
		sc.mu.RUnlock()
		return
	}
	sc.mu.RUnlock()
	//命令需要执行
	//尝试在raft层启动命令
	index, _, isLeader := sc.rf.Start(Command{args})
	if !isLeader {
```

```
        reply.Err = ErrWrongLeader
        return
    }
    sc.mu.Lock()
    //获取通知通道等待结果
    notifyChan := sc.getNotifyChan(index)
    sc.mu.Unlock()

    DPrintf("执行%v", args.Op)
    select {
    case result := <-notifyChan:
        DPrintf("客户端收到编号%v答复", result.Config.Num)
        reply.Config, reply.Err = result.Config, result.Err
    case <-time.After(ExecuteTimeout):
        reply.Err = ErrTimeout
    }

    go func() {
        sc.mu.Lock()
        //删除过时的通知通道以减少内存占用
        sc.removeOutdateNotifyChan(index)
        sc.mu.Unlock()
    }()
}
```

```
// getNotifyChan获取给定索引的通知通道,如果通道不存在，则创建一个
func (sc *ShardCtrler) getNotifyChan(index int) chan *CommandReply {
    notifyChans, ok := sc.notifyChans[index]
    if !ok {
        notifyChans = make(chan *CommandReply, 1)
        sc.notifyChans[index] = notifyChans
    }
    return notifyChans
}

// removeOutdateNotifyChan 移除通道
func (sc *ShardCtrler) removeOutdateNotifyChan(index int) {
    delete(sc.notifyChans, index)
}

// isDuplicateRequest 检查命令是否是重复命令
func (sc *ShardCtrler) isDuplicateRequest(clientId int64, commandId int64) bool
{
    OperationContext, ok := sc.LastOperations[clientId]
    return ok && commandId <= OperationContext.MaxAppliedCommandId
}
```

applier用于开启服务器监听raft集群的消息回复，并及时发送给处理RPC，使其返回客户端

```
// applier 应用程序是将日志应用到状态机的程序
func (sc *ShardCtrler) applier() {
    for !sc.Killed() {
        select {
        case message := <-sc.applyCh:
            // 处理 Raft 的日志条目
            if message.CommandValid {
```

```
                reply := new(CommandReply)
                command := message.Command.(Command)
                sc.mu.Lock()

                if command.Op != Query &&
sc.isDuplicateRequest(command.ClientId, command.CommandId) {
                    reply = sc.LastOperations[command.ClientId].LastReply
                } else {
                    reply = sc.applyLogToStateMachine(command)
                    if command.Op != Query {
                        sc.LastOperations[command.ClientId] = OperationContext{
                            MaxAppliedCommandId: command.CommandId,
                            LastReply:           reply,
                        }
                    }
                }
                //当节点为leader时，通知相关通道currentTerm的日志
                //通知 Leader 的客户端：确保客户端请求（如 Join、Leave）在日志提交并应用后
收到响应
                DPrintf("执行%s成功", command.Op)
                if currentTerm, isLeader := sc.rf.GetState(); isLeader &&
message.CommandTerm == currentTerm {
                    notifyChan := sc.getNotifyChan(message.CommandIndex)
                    notifyChan <- reply
                }
                sc.mu.Unlock()
            }
        }
    }
}
```

用于将已经被大多数raft集群提交的命令，复制到状态机。

```
func (sc *ShardCtrler) applyLogToStateMachine(command Command) *CommandReply {
    reply := new(CommandReply)
    switch command.Op {
    case Join:
        reply.Err = sc.stateMachine.Join(command.Servers)
    case Leave:
        reply.Err = sc.stateMachine.Leave(command.GIDs)
    case Move:
        reply.Err = sc.stateMachine.Move(command.Shard, command.GID)
    case Query:
        reply.Config, reply.Err = sc.stateMachine.Query(command.Num)
    }
    DPrintf("应用操作%s到状态机", command.Op)
    return reply
}
```

configStateMachine

定义接口类型，其中包括应用状态机的 4种操作

```go
//src/shardctrler/configStateMachine.go
type ConfigStateMachine interface {
	Join(group map[int][]string) Err //键是副本组的唯一标识符（GID，非零整数），值是该组中服务器的地址列表（例如 ["server1:port", "server2:port"]）。
	Leave(gids []int) Err
	Move(shard, gid int) Err
	Query(num int) (Config, Err)
}
```

实现该接口

```go
type MemoryConfigStateMachine struct {
	Configs []Config
}

func NewMemoryConfigStateMachine() *MemoryConfigStateMachine {
	cf := &MemoryConfigStateMachine{make([]Config, 1)}
	cf.Configs[0] = DefaultConfig()
	return cf
}
```

获取拥有最多/最少分片的组id。对于找寻最多分片组Id，我们优先处理组id为0，因为0中包含还为分片索引。对于找寻最少分片组Id，不需要考虑0

```go
// GetGidWithMinimumShards 返回具有最大分片数的组ID
func GetGidWithMinimumShards(group2Shards map[int][]int) int {
	//函数的目的是找到一个活跃的副本组（gid != 0），以接收从其他组（如 GID 0 或分片多的组）移动来的分片。
	// 选择分片最少的有效组（如 GID 2 有 2 个分片），确保分配后负载更均衡（如接近 NShards / len(Groups)）

	// 获得所有的组Id
	var gids []int
	for gid := range group2Shards {
		gids = append(gids, gid)
	}
	sort.Ints(gids)
	index, minShards := -1, NShards+1
	for _, gid := range gids {
		// 不考虑0组
		if gid != 0 && len(group2Shards[gid]) < minShards {
			index, minShards = gid, len(group2Shards[gid])
		}
	}
	return index
}

// GetGIDWithMaximumShards 返回具有最大分片数的组ID
func GetGidWithMaximumShards(group2Shards map[int][]int) int {
	//Group表示未分配的组，如果有未分配的分片，则选择gid 0
	if shards, ok := group2Shards[0]; ok && len(shards) != 0 {
		//初始配置：Shards = [0,0,...,0]，所有分片分配给 GID 0，表示系统启动时无有效组（Groups = {}）。
		//组id0中的分片  都是未分配组的分片，需要优先处理
		return 0
	}
```

```go
    var gids []int
    for gid := range group2Shards {
        gids = append(gids, gid)s
    }
    sort.Ints(gids)
    index, maxShards := -1, -1
    //查找具有最大分片数的组ID
    for _, gid := range gids {
        if len(group2Shards[gid]) > maxShards {
            index, maxShards = gid, len(group2Shards[gid])
        }
    }
    return index
}
```

一些功能函数

```go
// Group2Shards Group2Shards 将 Config.Shards（分片到 GID 的数组）转换为 map[int][]int（GID 到分片列表的映射）。
func Group2Shards(config Config) map[int][]int {
    group2Shards := make(map[int][]int)
    for gid := range config.Groups {
        group2Shards[gid] = make([]int, 0)
    }
    for shard, gid := range config.Shards {
        group2Shards[gid] = append(group2Shards[gid], shard)
    }
    return group2Shards
}

// deepCopy 创建一个组映射的  深拷贝副本
func deepCopy(groups map[int][]string) map[int][]string {
    newGroups := make(map[int][]string)
    for gids, servers := range groups {
        newServers := make([]string, len(servers))
        copy(newServers, servers)
        newGroups[gids] = newServers
    }
    return newGroups
}
```

状态机的四种命令实现

```go
// Join 添加新的组进入到配置中
func (cf *MemoryConfigStateMachine) Join(group map[int][]string) Err {
    DPrintf("状态机执开始行Join")
    lastConfig := cf.Configs[len(cf.Configs)-1]
    //基于最后一次的配置创建新的配置
    newConfig := Config{
        len(cf.Configs),
        lastConfig.Shards,
        deepCopy(lastConfig.Groups),
    }

    for gid, servers := range group {
        //如果group不在新配置里面，就添加
```

```go
        if _, ok := newConfig.Groups[gid]; !ok {
            newServers := make([]string, len(servers))
            copy(newServers, servers)
            newConfig.Groups[gid] = newServers
        }
    }
    group2Shards := Group2Shards(newConfig) //gid-shards
    for {
        // 对组之间的分片进行负载均衡
        source, target := GetGidWithMaximumShards(group2Shards),
GetGidWithMinimumShards(group2Shards)
        if source != 0 && len(group2Shards[source])-len(group2Shards[target]) <=
1 {
            break
        }
        group2Shards[target] = append(group2Shards[target], group2Shards[source]
[0])
        group2Shards[source] = group2Shards[source][1:]
    }
    //更新新配置中的分片分配
    var newShards [NShards]int
    for gid, shards := range group2Shards {
        for _, shard := range shards {
            newShards[shard] = gid
        }
    }

    newConfig.Shards = newShards
    cf.Configs = append(cf.Configs, newConfig)
    DPrintf("状态机执行Join完成")
    return OK

}

// Leave  从配置中删除指定的组
func (cf *MemoryConfigStateMachine) Leave(gids []int) Err {
    lastConfig := cf.Configs[len(cf.Configs)-1]
    newConfig := Config{
        len(cf.Configs),
        lastConfig.Shards,
        deepCopy(lastConfig.Groups),
    }

    group2Shards := Group2Shards(newConfig)
    //用于存储孤儿碎片
    orphanShards := make([]int, 0)
    for _, gid := range gids {
        //如果新配置中存在该组，则删除它
        if _, ok := newConfig.Groups[gid]; ok {
            delete(newConfig.Groups, gid)
        }
        // 删除gid -> shards的映射
        if shards, ok := group2Shards[gid]; ok {
            delete(group2Shards, gid)
            orphanShards = append(orphanShards, shards...)
        }
    }
```

```go
    var newShards [NShards]int
    if len(newConfig.Groups) > 0 {
        //将孤立碎片重新分配给剩余的组
        for _, shard := range orphanShards {
            gid := GetGidWithMinimumShards(group2Shards)
            newShards[shard] = gid
            group2Shards[gid] = append(group2Shards[gid], shard)
        }

        //在新配置中更新分片分配
        for gid, shard := range group2Shards {
            for _, shard := range shard {
                newShards[shard] = gid
            }
        }
    }
    newConfig.Shards = newShards
    cf.Configs = append(cf.Configs, newConfig)
    return OK
}

// Move 命令用于将指定的shard移动到指定的组中。
func (cf *MemoryConfigStateMachine) Move(shard, gid int) Err {
    lastConfig := cf.Configs[len(cf.Configs)-1]
    //根据上次配置创建新配置
    newConfig := Config{
        len(cf.Configs),
        lastConfig.Shards,
        lastConfig.Groups,
    }
    //根据上次配置创建新配置
    newConfig.Shards[shard] = gid
    cf.Configs = append(cf.Configs, newConfig)
    return OK
}

// 查询指定配置
func (cf *MemoryConfigStateMachine) Query(num int) (Config, Err) {
    //如果配置号无效，则返回最新的配置
    if num < 0 || num >= len(cf.Configs) {
        return cf.Configs[len(cf.Configs)-1], OK
    }
    return cf.Configs[num], OK
}
```

结果

```
mit2024/6.5840/src/shardctrler$ go test
Test: Basic leave/join ...
  ... Passed
Test: Historical queries ...
  ... Passed
Test: Move ...
  ... Passed
Test: Concurrent leave/join ...
  ... Passed
Test: Minimal transfers after joins ...
```

```
  ... Passed
Test: Minimal transfers after leaves ...
  ... Passed
Test: minimal movement again ...
  ... Passed
Test: Multi-group join/leave ...
  ... Passed
Test: Concurrent multi leave/join ...
  ... Passed
Test: Minimal transfers after multijoins ...
  ... Passed
Test: Minimal transfers after multileaves ...
  ... Passed
Test: Check Same config on servers ...
  ... Passed
PASS
ok      6.5840/shardctrler      9.118s
```