Lab2 KV服务

写在前面

在本实验中，您将构建一个单机键/值服务器，确保在发生网络故障时每个操作都能精确执行一次，并且操作能够线性化。后续的实验将复制这样的服务器以处理服务器崩溃。

客户端可以向键/值服务器发送三种不同的 RPC： `Put(key, value)` 、 `Append(key, arg)` 和 `Get(key)` 。服务器维护一个内存中的键/值对映射。键和值是字符串。 `Put(key, value)` 用于安装或替换映射中特定键的值， `Append(key, arg)` 将 arg 追加到键的值并返回旧值， `Get(key)` 获取键的当前值。对于不存在的键， `Get` 应返回一个空字符串。

对于不存在的键， `Append` 应将现有值视为空字符串。每个客户端通过一个具有 Put/Append/Get 方法的 `Clerk` 与服务器通信。一个 `Clerk` 负责与服务器进行 RPC 交互。

您的服务器必须确保应用程序对 `Clerk` Get/Put/Append 方法的调用能够实现线性化。如果客户端请求不是并发的，每个客户端的 Get/Put/Append 调用都应观察到由先前序列调用所暗示的状态修改。对于并发调用，返回值和最终状态必须与某些顺序一次执行这些操作时相同。调用是并发的，如果它们在时间上重叠：例如，如果客户端 X 调用 `Clerk.Put()` ，客户端 Y 调用 `Clerk.Append()` ，然后客户端 X 的调用返回。调用必须观察到在调用开始之前已完成的所有调用的效果。

线性化对于应用程序来说很方便，因为它就像一个一次处理一个请求的单个服务器的行为。例如，如果一个客户端从服务器成功收到了一个更新请求的响应，随后其他客户端发起的读取操作都会保证能看到该更新的效果。为单个服务器提供线性化相对容易。

开始使用

我们为您提供了一个骨架代码和测试用例 `src/kvsrv` 。您需要修改 `kvsrv/client.go` 、 `kvsrv/server.go` 和 `kvsrv/common.go` 。要启动运行，请执行以下命令。别忘了使用 `git pull` 获取最新软件。

无网络故障的键值服务器（简单）

1. 您的第一个任务是实现一个解决方案，当没有丢失的消息时能够正常工作。
2. 你需要在 `client.go` 中向 Clerk 的 Put/Append/Get 方法中添加 RPC 发送代码，并在 `server.go` 中实现 `Put` 、 `Append()` 和 `Get()` 的 RPC 处理程序。
3. 当您通过测试套件中的前两个测试“一个客户端”和“多个客户端”时，即完成了此任务。

任务要求

此任务需要实现一个在没

有丢失消息的情况下有效的解决方案。你需要在 client.go 中，在 Clerk 的 Put/Append/Get 方法中添加 RPC 的发送代码；并且实现 server.go 中 Put、Append、Get 三个 RPC handler。

Server

```
//src/kvsrv/server.go
package kvsrv

import (
    "log"
```

```
    "sync"
)

const Debug = false

func DPrintf(format string, a ...interface{}) (n int, err error) {
    if Debug {
        log.Printf(format, a...)
    }
    return
}

type KVServer struct {
    mu   sync.Mutex
    data map[string]string
    // Your definitions here.
}

func (kv *KVServer) Get(args *GetArgs, reply *GetReply) {
    // Your code here.
    kv.mu.Lock()
    defer kv.mu.Unlock()
    reply.Value = kv.data[args.Key]
}

func (kv *KVServer) Put(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    kv.mu.Lock()
    defer kv.mu.Unlock()
    kv.data[args.Key] = args.Value
}

func (kv *KVServer) Append(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    kv.mu.Lock()
    defer kv.mu.Unlock()
    oldValue := kv.data[args.Key]
    kv.data[args.Key] = oldValue + args.Value
    reply.Value = oldValue
}

func StartKVServer() *KVServer {
    kv := new(KVServer)
    kv.data = make(map[string]string)
    InitLogger()
    // You may need initialization code here.

    return kv
}
```

client

```
//src/kvsrv/client.go
package kvsrv

import (
    "crypto/rand"
```

```
    "math/big"

    "6.5840/labrpc"
)

type Clerk struct {
    server *labrpc.ClientEnd
    // You will have to modify this struct.
}

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

func MakeClerk(server *labrpc.ClientEnd) *Clerk {
    ck := new(Clerk)
    ck.server = server
    InitLogger()
    // You'll have to add code here.
    return ck
}

// fetch the current value for a key.
// returns "" if the key does not exist.
// keeps trying forever in the face of all other errors.
//
// you can send an RPC with code like this:
// ok := ck.server.Call("KVServer.Get", &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) Get(key string) string {
    args := GetArgs{
        Key: key,
    }
    reply := GetReply{}
    if ok := ck.server.Call("KVServer.Get", &args, &reply); !ok {
        SugarLogger.Info("客户端调用Get方法失败")
        return ""
    }
    // You will have to modify this function.
    return reply.Value
}

// shared by Put and Append.
//
// you can send an RPC with code like this:
// ok := ck.server.Call("KVServer."+op, &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) string {
    // You will have to modify this function.
```

```
    args := PutAppendArgs{
        Key:   key,
        Value: value,
    }
    reply := PutAppendReply{}
    SugarLogger.Infof("客户端开始调用KVServer.%s", op)
    if ok := ck.server.Call("KVServer."+op, &args, &reply); !ok {
        SugarLogger.Infof("客户端调用KVServer.%s失败", op)
        return ""
    }
    SugarLogger.Infof("客户端调用KVServer.%s成功", op)
    return reply.Value
}

func (ck *Clerk) Put(key string, value string) {
    ck.PutAppend(key, value, "Put")
}

// Append value to key's value and return that value
func (ck *Clerk) Append(key string, value string) string {
    return ck.PutAppend(key, value, "Append")
}
```

带丢弃消息功能的键/值服务器

现在你应该修改你的解决方案，使其能够在消息丢失（例如，RPC 请求和 RPC 回复）的情况下继续运行。如果一条消息丢失了，那么客户端的 `ck.server.Call()` 将返回 `false` （更精确地说，`Call()` 会在超时时间内等待回复消息，如果没有在该时间内收到回复，则返回 false）。你将面临的一个问题是，一个 `Clerk` 可能需要多次发送 RPC 请求直到成功。然而，每次对 `Clerk.Put()` 或 `Clerk.Append()` 的调用都应该只执行一次，因此你必须确保重发不会导致服务器重复执行请求。

任务：

在 `Clerk` 中添加代码，如果未收到回复则重试，并在需要时在 `server.go` 中添加代码过滤重复项。这些注释包括重复检测的指导。

提示：

1. 您需要唯一标识客户端操作，以确保键/值服务器每次只执行一次。
2. 您需要仔细考虑服务器必须维护什么状态来处理重复的 `Get()` 、 `Put()` 和 `Append()` 请求，如果有的话。
3. 您的重复检测方案应快速释放服务器内存，例如，每次 RPC 都意味着客户端已经看到了其前一个 RPC 的回复。可以假设客户端在同一时间只会调用一次 Clerk。

Common

```
//src/kvsrv/common.go
type MessageType int

const (
    Modify = iota   //修改状态
```

```
    Report //任务完成，报告状态
)

// Put or Append
type PutAppendArgs struct {
    Key         string
    Value       string
    MessageType MessageType
    MessageID   int64
    // You'll have to add definitions here.
    // Field names must start with capital letters,
    // otherwise RPC will break.
}
```

这里设置MesssageType 和 MessageID 。其中MesssageType 有Modify和Report两个类型，分别表示任务未完成，和任务完成状态

MessageID 用来为每个ID创建唯一的一个标识符。

Server

由于Get函数只涉及读取，所以当重复请求到来的时候，Get获取当前最新的值。

Put和Append进行重试和幂等性机制

```
//src/kvsrv/server.go
func (kv *KVServer) Put(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    if args.MessageType == Report { //此请求是针对已完成的操作，用于通知服务器需要移除相关数据。
        kv.record.Delete(args.MessageID)
        return
    }
    res, ok := kv.record.Load(args.MessageID)
    if ok {
        reply.Value = res.(string) // 重复请求，返回之前的结果
        return
    }
    // 非重复请求
    kv.mu.Lock()
    ordValue := kv.data[args.Key] //旧值需要存入map中，再该请求没有被确认之前，返回的一定是之前的旧值重复请求可直接返回之前的缓存
    kv.data[args.Key] = args.Value
    reply.Value = ordValue
    kv.mu.Unlock()
    kv.record.Store(args.MessageID, ordValue)
}
```

Append操作和Put操作类似

```
// //src/kvsrv/server.go
func (kv *KVServer) Append(args *PutAppendArgs, reply *PutAppendReply) {
    // Your code here.
    if args.MessageID == Report {
        kv.record.Delete(args.MessageID)
        return
    }
```

```
    res, ok := kv.record.Load(args.MessageID)
    if ok { //重复请求，返回缓存的结果
        reply.Value = res.(string)
        return
    }
    kv.mu.Lock()
    oldValue := kv.data[args.Key]
    kv.data[args.Key] = oldValue + args.Value
    reply.Value = oldValue
    kv.mu.Unlock()
    kv.record.Store(args.MessageID, oldValue)
}
```

Client

客户端则就说无限的调用rpc一直到任务完成，任务完成后需要告知服务器该任务完成，让服务释放缓存。

```
// 设置幂等性机制
func (ck *Clerk) PutAppend(key string, value string, op string) string {
    // You will have to modify this function.
    messageID := nrand()
    args := PutAppendArgs{
        Key:         key,
        Value:       value,
        MessageType: Modify,
        MessageID:   messageID,
    }
    reply := PutAppendReply{}
    for !ck.server.Call("KVServer."+op, &args, &reply) {
    }
    args = PutAppendArgs{
        MessageType: Report,
        MessageID:   messageID,
    }
    for !ck.server.Call("KVServer."+op, &args, &reply) {
    }

    return reply.Value
}
```

结果

```
xy@xy:~/mit2024/6.5840/src/kvsrv$ go test
Test: one client ...
labgob warning: Decoding into a non-default variable/field Value may not work
  ... Passed --  t  3.6 nrpc 47721 ops 31821
Test: many clients ...
info: linearizability check timed out, assuming history is ok
  ... Passed --  t  5.5 nrpc 165215 ops 110211
Test: unreliable net, many clients ...
  ... Passed --  t  3.3 nrpc  1103 ops  596
Test: concurrent append to same key, unreliable ...
  ... Passed --  t  0.5 nrpc   125 ops   52
Test: memory use get ...
  ... Passed --  t  0.4 nrpc     8 ops    0
```

```
Test: memory use put ...
  ... Passed --  t  0.2 nrpc     4 ops    0
Test: memory use append ...
  ... Passed --  t  0.3 nrpc     4 ops    0
Test: memory use many put clients ...
  ... Passed --  t 15.0 nrpc 200000 ops    0
Test: memory use many get client ...
  ... Passed --  t 10.8 nrpc 100002 ops    0
Test: memory use many appends ...
2025/03/05 15:49:23 m0 658128 m1 1667408
  ... Passed --  t  1.6 nrpc  2000 ops    0
PASS
ok      6.5840/kvsrv    43.292s
```