

## gemini-conversation

Exported on: 12/7/2025, 10:29:28 PM

---

**User**

I have attached my code below for a leader. I am trying to implement input rate limiting but am struggling on to how to do this efficiently. Here is my code so far:

```

package main

import (
    "bufio"
    "fmt"
    "g14-mp4/RainStorm/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "strconv"
    "strings"
    "sync"
    "time"
)

type WorkerIps struct {
    ips []net.IP
    sync.RWMutex
}

type RainStorm struct {
    NumStages int
    NumTasksPerStage int
    HydfsSrcDirectory string
    HydfsDestinationFileName string
    ExactlyOnce bool
    AutoScale bool
    InputRate float64
    LowestRate float64
    HighestRate float64
    Ops []Operation
    Ips map[int]net.IP
    taskMap map[int]map[int]net.IP
    stageMap map[int]map[int]net.IP
    NextTaskNum map[int]int
    NextAvailableVM int
    Lock *sync.Mutex
}

var workers WorkerIps
var numWorkers int
var numSuccessfulDials int
var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex
var appCompletedChan chan bool
var dataDir string

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}

    appCompletedChan = make(chan bool, 1)
    go func() {
        err := rpc.Register(&workers)
        if err != nil {
            fmt.Println(err)
            return
        }
        listener, err := net.Listen("tcp", "IntroducePort")
        if err != nil {
            fmt.Println(err)
            return
        }
        rpc.Accept(listener)
    }()
    input := make(chan RainStorm)
    go processStdin(input)
    hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
    if err != nil {
        fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error())
    }
    defer hydfsClient.Close()

    for {
        r := <-input // INITIATE NEW RAINSTORM APPLICATION
        workers.Lock()
        rpcWorkers = make(map[string]*rpc.Client)
        numSuccessfulDials = 0
        rpcWorkersLock.Lock()
        for _, workerIp := range workers.Ips {
            //collect list of tasks for this worker
            err := rpc.Dial("tcp", workerIp.String() + AssignmentPort)
            if err != nil {
                fmt.Println("Unable to connect to worker: " + err.Error())
                continue
            }
            rpcWorkers[workerIp.String()] = worker
            numSuccessfulDials++
        }
        rpcWorkersLock.Unlock()
        r.Lock = new(sync.Mutex)
        r.Lock.Lock()
        r.Ips = make([]net.IP, r.NumStages)
        r.NextTaskNum = make([]int, r.NumStages)
        r.TaskCompletion = make([]CompletionTuple, r.NumStages)
        r.initWorker()
        r.NextAvailableVM = 0
        for i := range r.NumStages {
            r.Ips[i] = make(map[int]net.IP)
            r.TaskCompletion[i] = CompletionTuple{Counter: 0}
        }
        for _ = range r.NumTasksPerStage {
            r.addTask(i)
        }
        workers.Unlock()
        r.sendIps()
        r.Lock.Unlock()
    }
}

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3. compare rates to see if changes are needed 4. complete changes */
appServer := rpc.NewServer()
err = appServer.Register(r)
if err != nil {
    fmt.Println(err)
    continue
}
globalRmListener, err := net.Listen("tcp", "GlobalRMPort")
if err != nil {
    fmt.Println("GlobalRM unable to connect to worker: " + err.Error())
    continue
}
go appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is done once done reading from the file // read in from local; output on HyDFS inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))
if err != nil {
    fmt.Println("Unable to open src directory: " + err.Error())
}
scanner := bufio.NewScanner(inputFile)
interval := time.Duration(float64(time.Second) / r.InputRate)
ticker := time.NewTicker(interval)
lineNum := 0

go func() {
    tupleClients := make(map[string]*WorkerClient, numWorkers)
    for range ticker.C {
        if !scanner.Scan() {
            //EOF @TODO break
            _, err := fmt.Fprintf(client.conn, "%s-%d,%d,%d,%s\n", out.taskId.String(), out.tupleId, nextStage, nextTask, out.output)
            if err != nil {
                fmt.Println("Error writing to client: " + err.Error())
            }
            scanner.Close()
            break
        }
        lineNum++
        if _, exists := tupleClients[out.tupleId]; !exists {
            tupleClients[out.tupleId] = &WorkerClient{
                taskId:    out.taskId,
                tupleId:   out.tupleId,
                stage:     nextStage,
                task:      nextTask,
                output:    out.output,
                numWorkers: numWorkers,
            }
        }
        tupleClients[out.tupleId].InputRate += float64(1) / interval.Nanoseconds()
        if tupleClients[out.tupleId].InputRate > r.InputRate {
            tupleClients[out.tupleId].InputRate = r.InputRate
        }
    }
}

// CLEANUP: do once the current RainStorm application is done
rpcWorkersLock.Lock()
for _, worker := range rpcWorkers {
    worker.Close()
}
rpcWorkersLock.Unlock()
err = globalRmListener.Close()
if err != nil {
    fmt.Println(err)
}
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error {
    workers.Lock()
    defer workers.Unlock()
    workers.Ips = append(workers.Ips, args)
    numWorkers++
    return nil
}

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    // restart the task on the next worker in the cycle
    app.Lock()
    defer app.Lock.Unlock()
    if _, exists := app.Ips[task.Stage][task.TaskNumber]; !exists {
        return nil
    }
    app.Ips[task.Stage][task.TaskNumber] = make(net.IP)
}
```

```

fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ app.sendIps() } return nil } func (app *RainStorm)
ReceiveRateUpdate(args RmUpdate, reply *int) error { app.Lock.Lock() defer app.Lock.Unlock() if
app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.addTask(args.Stage)
app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[args.Stage][args.TaskNumber]; exists { delete(app.ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}
} else { //do nothing because this should never happen fmt.Println("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.ips[stageNum] = append(app.ips[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.ips[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.l.RLock() app.ips[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ task := Task{ TaskNumber: taskNum, Stage:
stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.ips[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.ips[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.ips[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.ips[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.ips[stageNum], taskNum) app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task") }

```

```

task: " + err.Error() } }

func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case "RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch { case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args: splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i] break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i] break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: // AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true } break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": // @TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

This is a classic Distributed Systems challenge. You are currently using a `time.Ticker` which triggers on every single event.

The problem with this approach is **Granularity** and **Overhead**.

- CPU Overhead:** If your `InputRate` is high (e.g., 5,000 events/sec), the ticker wakes up the CPU every 200 microseconds. The overhead of context switching and channel management will eventually consume more CPU than the actual work of sending the tuple.
- Drift:** `time.Ticker` tries to keep up, but if your loop logic (scanning, parsing, RPC sending) takes longer than the tick interval, you will fall behind, and standard tickers drop ticks to catch up, leading to data loss or bursty behavior.

Here is a more efficient approach using **Batched Drift-Correction**.

## The Solution: Batched Drift-Correction

Instead of sleeping for every message, we process a small batch of messages (e.g., 10 or 100), calculate how much time *should* have passed, and then sleep for the remainder. This amortizes the cost of the system call to `time.Sleep`.

## Updated Code Implementation

Replace your current `go func()` with the implementation below. This logic dynamically adjusts; if the rate is low, it behaves like a ticker. If the rate is high, it processes in bursts to maintain high throughput without CPU thrashing.

Go



```
go func() {
    // 1. Setup Tracking Variables
    startTime := time.Now()
    processedCount := 0

    // We update our sleep timer every 'batchSize' records.
    // For high rates, a larger batch reduces CPU overhead.
    batchSize := 1
    if r.InputRate > 100 {
        batchSize = 20
    }
    if r.InputRate > 1000 {
        batchSize = 100
    }

    tupleClients := make(map[string]*WorkerClient, numWorkers) // Assuming WorkerClient is

    for scanner.Scan() {
        line := scanner.Text() // Get the data
        lineNum++

        // --- 2. YOUR SENDING LOGIC HERE ---
        // Parse the line and send it via RPC to the correct worker.
        // (Your original code had this commented out or missing,
        // make sure to hash the key to find the correct worker IP).
        // Example logic:
        // taskID := getHash(line) % r.NumTasksPerStage
        // workerIP := r.Ips[0][taskID]
        // sendTuple(workerIP, line)
        // ----

        processedCount++

        // 3. Rate Limiting Logic (The Efficient Part)
        if processedCount % batchSize == 0 {
            // Calculate where we SHOULD be in time based on the rate
            expectedDuration := time.Duration(float64(processedCount) / r.InputRate) * float

            // Calculate actual time elapsed
            actualElapsed := time.Since(startTime)

            // If we are ahead of schedule (too fast), sleep the difference
            if actualElapsed < expectedDuration {
                time.Sleep(expectedDuration - actualElapsed)
            }
            // If actualElapsed > expectedDuration, we are behind schedule.
        }
    }
}
```

```
// We do NOT sleep, effectively allowing the system to "catch up".
}

}

if err := scanner.Err(); err != nil {
    fmt.Println("Error reading input file:", err)
}

_ = inputFile.Close()

// Signal completion of reading source file
// Note: You likely need a mechanism here to tell the system
// that the *Source* stage is done, sending a special "End of Stream" message.
}()


```

## Why this works better for RainStorm:

1. **Precision:** It calculates sleep based on `(TotalItems / Rate)`. This mathematically guarantees that over a 10-second period, you will have sent exactly `10 * Rate` items, correcting for any minor processing delays automatically.
2. **Efficiency:** By using `batchSize`, you perform the expensive `time.Now()` and `time.Sleep()` calls significantly fewer times.
3. **Smoothness:** It handles the "Catch Up" naturally. If the Garbage Collector runs and pauses your app for 5ms, the `actualElapsed` will be higher than `expectedDuration`, so the loop will skip the `Sleep` block until it catches up to the target rate.

## Next Step for you

You currently have `tupleClients := make(map[string]*WorkerClient, numWorkers)` inside the routine but you aren't dialing or using them yet.

Would you like me to generate the **Source Hash Partitioning** logic to fill in the `// --- 2. YOUR SENDING LOGIC HERE ---` section, ensuring the input lines are correctly distributed to your Stage 1 workers?

**User**

but there is also autscaling, so what about the case where the first stage gets auto scaled while the input source is still being sent

**Gemini**

This is a critical distributed systems edge case. If you autoscale (add/remove tasks) or handle failures while the input loop is running, you introduce two major risks:

1. **Race Condition Panic:** Your input loop is reading `r.Ips` while `ReceiveRateUpdate` is writing to it. In Go, concurrent map read/writes cause a fatal crash.
2. **Stale Routing:** You might send data to a worker that no longer owns that task (or has crashed).

To fix this **efficiently** (without locking on every single tuple), you should use the **Local Snapshot Pattern**.

**The Solution: Local Snapshotting**

The input loop will hold its own *local copy* of the Stage 1 routing table. It only locks the global mutex to refresh this copy when it receives a signal that the topology has changed.

Here is the implementation:

**1. Add a Signal Channel**

Add a buffered channel to your struct to signal when Stage 1 changes.

Go



```
type RainStorm struct {
    // ... existing fields ...
    Stage1UpdateChan chan bool // Add this
}

// Initialize this in main() before starting the processor
// r.Stage1UpdateChan = make(chan bool, 10)
```

**2. Update the Input Loop (The Reader)**

Modify the `go func` I gave you previously. Inside the loop, we check if an update is pending. If yes, we pause briefly to refresh our routing table.

Go



```
go func() {
    // 1. Initial Snapshot of Stage 1
    // We make a local copy so we don't need to lock r.Lock on every single tuple
    r.Lock.Lock()
    stage1Map := make(map[int]string) // Local cache: TaskID -> WorkerIP
    for taskID, ip := range r.Ips[0] {
        stage1Map[taskID] = ip.String()
    }
}
```

```
r.Lock.Unlock()

// Setup Clients Cache
// Map of "IP:Port" -> RPC Client
workerClients := make(map[string]*rpc.Client)
// (Helper function to connect/get client needed here)

// ... Batching logic setup (from previous answer) ...

for scanner.Scan() {
    // --- A. CHECK FOR TOPOLOGY UPDATES ---
    // We use a non-blocking select. If no update, we continue immediately.
    select {
        case <-r.Stage1UpdateChan:
            r.Lock.Lock()
            // Refresh our local snapshot
            stage1Map = make(map[int]string)
            for taskID, ip := range r.Ips[0] {
                stage1Map[taskID] = ip.String()
            }
            r.Lock.Unlock()
            fmt.Println("Input source detected Stage 1 topology change. Routing updated.")
        default:
            // No changes, keep processing
    }

    // --- B. ROUTING LOGIC ---
    line := scanner.Text()
    lineNum++

    // 1. Determine which TaskID handles this tuple (Partitioning)
    // Usually Hash(Line) % NumTasks
    hash := 0
    for i := 0; i < len(line); i++ {
        hash = 31*hash + int(line[i])
    }
    if hash < 0 { hash = -hash } // Handle negative overflow
    targetTaskID := hash % r.NumTasksPerStage

    // 2. Look up the IP in our LOCAL snapshot (No locking needed!)
    targetIP, exists := stage1Map[targetTaskID]
    if !exists {
        // This happens if a task was removed but we haven't rescaled inputs yet
        // Just drop, or round-robin to another available task
        continue
    }
}
```

```

    // 3. Send Tuple (using cached RPC connection)
    // client := getClient(workerClients, targetIP)
    // client.Call("Worker.ProcessTuple", ...)

    processedCount++

    // --- C. BATCH RATE LIMITING (From previous answer) ---
    if processedCount % batchSize == 0 {
        // ... logic from previous step ...
    }
}

// ... cleanup ...
}()
```

### 3. Update the Triggers (The Writers)

You must notify the channel whenever you modify `r.Ips[0]` (Stage 1). You need to add this notification logic to `ReceiveFailure`, `ReceiveRateUpdate` (specifically inside `addTask` and `removeTask`).

In `addTask`:

```

Go □

func (app *RainStorm) addTask(stageNum int) {
    // ... existing logic ...

    // AFTER updating app.Ips, notify if it was Stage 0
    if stageNum == 0 {
        // Non-blocking send (prevent deadlock if channel is full)
        select {
        case app.Stage1UpdateChan <- true:
        default:
        }
    }
}
```

In `removeTask`:

```

Go □

func (app *RainStorm) removeTask(stageNum int) {
    // ... existing logic ...

    delete(app.Ips[stageNum], taskNum)

    // AFTER updating app.Ips, notify if it was Stage 0
```

```

if stageNum == 0 {
    select {
        case app.Stage1UpdateChan <- true:
        default:
    }
}
// ... existing logic ...
}

```

In `ReceiveFailure`:

Go



```

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    // ... existing locking ...

    app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]

    // Notify input loop if the failed task was in Stage 0
    if task.Stage == 0 {
        select {
            case app.Stage1UpdateChan <- true:
            default:
        }
    }

    // ... existing logic ...
}

```

**Why this is the "Leader" choice:**

- Zero-Cost Reads:** The input loop does 99% of the work. By using a local map `stage1Map`, you avoid `r.Lock.Lock()` entirely for the hot path.
- Thread Safety:** You only touch the global `r.Ips` map when the `select` case triggers, and you wrap that specific access in a Lock.
- Responsiveness:** As soon as `addTask` finishes, the very next tuple read from the file will pick up the change.

**Next Step**

You are using `rpc.Dial` inside loops or relying on `rpcWorkers` map. Would you like me to write a helper function `getRpcClient(ip string)` that manages a connection pool for the input loop, ensuring you don't run out of file descriptors (sockets) by dialing too many times?

**User**

do you not need to explicitly make the channel?

**Gemini**

Yes, you **absolutely must** explicitly `make` the channel.

In Go, if you define a channel field in a struct but don't `make` it, it remains `nil`.

- **Sending** to a `nil` channel blocks forever (deadlocking your `addTask` or `ReceiveFailure`).
- **Receiving** from a `nil` channel blocks forever (or in a `select` statement, the case is ignored entirely, so your input loop would never update).

You need to initialize it **inside your main loop**, right where you initialize your Mutexes.

## 1. Update the Struct

First, add the field to your struct definition:

```
Go   
type RainStorm struct {  
    // ... existing fields ...  
    Lock          *sync.Mutex  
    Stage1UpdateChan chan bool // <---- Add this  
}
```

## 2. Update `main()`

In your `main` function, look for the section where you receive `r := <-input`. You are already initializing `r.Lock` there. Initialize the channel right next to it.

```
Go   
  
// Inside main()  
for {  
    r := <-input  
  
    // ... existing networking setup ...  
  
    r.Lock = new(sync.Mutex)  
    r.Lock.Lock()  
  
    // --- INSERT THIS LINE ---  
    // Buffer of 1 is sufficient to prevent the sender from blocking  
    // if the reader is busy, but a few more doesn't hurt.  
    r.Stage1UpdateChan = make(chan bool, 10)  
    // -----  
  
    r.Ips = make([]map[int]net.IP, r.NumStages)
```

```
r.NextTaskNum = make([]int, r.NumStages)

// ... rest of your code ...

}
```

## Why a buffer size of 10?

I recommend `make(chan bool, 10)` (buffered) rather than `make(chan bool)` (unbuffered).

- **Unbuffered:** The sender (e.g., `addTask`) must wait until the receiver (the input loop) reads the value. If the input loop is sleeping or busy processing a large file chunk, `addTask` pauses.
- **Buffered:** `addTask` drops the signal in the "mailbox" and returns immediately. The input loop picks it up on its next cycle. This prevents your control plane (RPCs) from stuttering just because the data plane (Input processing) is busy.

## Next Step

You currently have this TODO in your code: `// needs to wait for the application to complete before cleaning up --> @T0D0: come back to this`

Right now, your code blocks on `<-appCompletedChan`. Would you like me to implement the **Ack-Tree** or **Coordinator** logic required to actually send that signal to `appCompletedChan` so your leader knows when the job is done?

**User**

I do not want to use batching, but I do like the idea of using the snapshotting technique

**Gemini**

This is a great choice. The **Local Snapshot** pattern solves the race conditions, and by removing batching, your code becomes conceptually simpler (process one line, send one line).

To handle the rate limiting efficiently without batching, we will use **Cumulative Drift Correction**.

**How Cumulative Drift Correction Works (No Batching)**

Instead of calculating how long to sleep for *this specific item* (which accumulates error), we calculate **when the Nth item was supposed to be sent**.

1. **Start Time:**  $T_0$

2. **Item N Target Time:**  $T_0 + (N / \text{Rate})$

3. **Current Time:**  $T_{\text{now}}$

4. **Action:**

- If  $T_{\text{now}} < \text{Target}$ , we are too fast. **Sleep** the difference.
- If  $T_{\text{now}} > \text{Target}$ , we are behind. **Do not sleep** (catch up immediately).

This is mathematically perfect over time. Even if one sleep call oversleeps by 1ms, the next iteration sees we are behind and skips sleeping, correcting the drift instantly.

**The Implementation**

Here is the complete replacement for your `go func()` loop. It combines the **Snapshotting** (for thread-safe autoscaling) and **Drift Correction** (for rate limiting).

Go



```
go func() {
    // 1. SETUP: Local Snapshot of Stage 1
    // We create a local map so we don't need to lock the main mutex on every tuple.
    r.Lock.Lock()
    stage1Map := make(map[int]string)
    for taskID, ip := range r.Ips[0] {
        stage1Map[taskID] = ip.String()
    }
    r.Lock.Unlock()

    // 2. SETUP: Connection Pool
    // We reuse connections to avoid running out of ports (Ephemeral Port Exhaustion).
    // Map key: "IP:Port", Value: *rpc.Client
    clientPool := make(map[string]*rpc.Client)

    // 3. SETUP: Rate Limiting
    startTime := time.Now()
    var processedCount float64 = 0
}
```

```
// 4. MAIN LOOP
for scanner.Scan() {
    line := scanner.Text()
    processedCount++

    // --- A. SNAPSHOT UPDATE (Non-blocking) ---
    // Check if the leader signaled a topology change
    select {
        case <-r.Stage1UpdateChan:
            r.Lock.Lock()
            // Refresh the local map completely
            stage1Map = make(map[int]string)
            for taskID, ip := range r.Ips[0] {
                stage1Map[taskID] = ip.String()
            }
            r.Lock.Unlock()
            // Clear connection pool (optional, or just let them lazily reconnect)
            // clientPool = make(map[string]*rpc.Client)
        default:
            // No update, keep going
    }

    // --- B. ROUTING (Hash Partitioning) ---
    // Deterministically assign this line to a Task ID
    hash := 0
    for i := 0; i < len(line); i++ {
        hash = 31*hash + int(line[i])
    }
    if hash < 0 { hash = -hash }

    targetTaskID := hash % r.NumTasksPerStage

    // Lookup IP in our safe LOCAL snapshot
    targetIP, exists := stage1Map[targetTaskID]
    if !exists {
        // Task might be temporarily missing during a rescale
        // Drop it, or log it.
        continue
    }

    // --- C. SENDING (With Connection Pooling) ---
    address := targetIP + AssignmentPort // e.g., "192.168.1.5:8000"
    client, ok := clientPool[address]
    if !ok {
        // Establish new connection if we don't have one
    }
}
```

```

        var err error
        client, err = rpc.Dial("tcp", address)
        if err != nil {
            fmt.Println("Error connecting to worker:", err)
            // If connection failed, remove from map so we retry next time
            delete(clientPool, address)
            continue
        }
        clientPool[address] = client
    }

    // Define your Tuple struct here or use your existing one
    // tuple := Tuple{Key: ..., Value: line}
    // var reply int
    // callErr := client.Call("Worker.ProcessTuple", tuple, &reply)
    // if callErr == rpc.ErrShutdown { delete(clientPool, address) }

    // --- D. RATE LIMITING (Drift Correction) ---
    // Calculate exactly when this Nth tuple SHOULD have been sent
    expectedDuration := time.Duration((processedCount / r.InputRate) * float64(time.Second))
    targetTime := startTime.Add(expectedDuration)

    now := time.Now()
    if targetTime.After(now) {
        // We are ahead of schedule. Sleep perfectly to sync up.
        time.Sleep(targetTime.Sub(now))
    }
    // If targetTime is before now, we are lagging. Loop immediately to catch up.
}

// Cleanup
if err := scanner.Err(); err != nil {
    fmt.Println("Error reading file:", err)
}
inputFile.Close()

// Optional: Send "End of Stream" markers here if needed
}()

```

## Critical implementation details to note

### 1. The "Wait" Logic

The logic `if targetTime.After(now)` is the key.

- If your system pauses for Garbage Collection for 10ms, `now` will jump forward.
- On the next loop, `targetTime` will likely be *before* `now`.

- The code skips `time.Sleep`, essentially bursting packets as fast as the CPU allows until it catches up to the average rate.

**2. The Connection Pool ( `clientPool` )** You cannot perform `rpc.Dial` inside this loop without checking if a connection exists.

- `rpc.Dial` performs a TCP Handshake (SYN, SYN-ACK, ACK).



- Doing this 100 times a second will introduce massive latency and eventually cause the OS to run out of sockets ("Too many open files").
- The `clientPool` map solves this by holding the open TCP connection and reusing it.

## Next Step

You are currently routing based on `hash(line)`. In many Stream Processing frameworks (like Storm or Flink), the **First Stage** often requires specific partitioning (e.g., splitting a sentence into words *before* hashing, or simply Round-Robining lines to distribute load evenly).

Do you want to stick with **Hash Partitioning** (sending identical lines to the same worker), or would you prefer **Round Robin** (sending line 1 to Worker A, line 2 to Worker B) for the input source to ensure perfect load balancing?

**User**

why would the drifting occur since we removed the use of the locks

**Gemini**

This is an excellent question. You are correct that removing the locks eliminates **contention** (waiting for other threads), but it does not eliminate **execution time** or **jitter**.

Even with the lock-free Snapshot technique, "Drift" occurs because **your code takes non-zero time to run**, and `time.Sleep` is not perfectly precise.

Here are the three reasons why drift still happens:

## 1. The "Work + Sleep" Loop Problem

If you target a rate of 1 message every 10ms, a naive loop looks like this:

1. **Send Message** (Takes 0.5ms to hash, lookup, and write to socket)
2. **Sleep(10ms)**

**Result:** One cycle actually takes **10.5ms**. After 1,000 messages, you are behind by **500ms** (0.5s). That is massive drift.

The Cumulative/Drift-Correction math fixes this by calculating: `Sleep Amount = Target Interval - Time Spent Working` (e.g., `Sleep(9.5ms)` instead of `Sleep(10ms)` ).

## 2. Go Garbage Collection (GC)

Go is a garbage-collected language. Every so often, the Go runtime pauses your program for a few microseconds or milliseconds to clean up memory (especially since you are creating strings with `scanner.Text()` in a loop).



- **Scenario:** You are perfectly on time.
- **Event:** GC Pauses the world for 2ms.
- **Result:** When your program wakes up, you are instantly 2ms behind schedule.

Without drift correction, you would just continue 2ms late forever. With drift correction, the next loop sees you are behind and skips `time.Sleep` to catch up.

## 3. `time.Sleep` Inaccuracy

`time.Sleep` is not a guarantee; it is a request to the OS scheduler.

- You ask: `time.Sleep(1ms)`
- The OS says: "Okay, I'll wake you up in at least 1ms."
- Result: The OS might wake you up in 1.2ms or 2ms depending on what other processes are running on the CPU.

## Summary

Removing the locks fixed the **Race Conditions** and **Deadlocks**, but the **Cumulative Math** is required to fix the **Physics of Time** (execution latency and OS scheduling).

Here is the breakdown of why the combination works:

- **Snapshotting:** Ensures `Send Message` doesn't crash or block for seconds.
- **Cumulative Math:** Ensures `Send Message + Sleep` always averages out to exactly your target Rate.

Would you like me to help you integrate the `getRpcClient` helper function now to ensure your `clientPool` logic is robust?

**User**

how would updates be sent to the channel

**Gemini**

You need to send the signal **immediately after you modify the `app.Ips` map** inside your Leader functions.

However, you must be careful: **Input operations are critical path**. You generally only need to notify the input loop if **Stage 0 (the first stage)** changes. If Stage 3 scales up, the file reader doesn't care—it just keeps sending to Stage 0.

Here is the best way to implement this using a helper method to ensure it never blocks your Leader.

## 1. Create a Helper Method

Add this method to `RainStorm`. It uses a `select` with a `default` case. This ensures that if the channel is already full (a signal is pending), the code **won't block/freeze** waiting for the input loop to wake up.

Go



```
func (app *RainStorm) notifyInputLoop(stageNum int) {
    // Only the input loop cares about Stage 0
    if stageNum != 0 {
        return
    }

    // Non-blocking send
    select {
        case app.Stage1UpdateChan <- true:
            // Signal sent successfully
        default:
            // Channel is full.
            // This is GOOD. It means a signal is already waiting to be read.
            // We don't need to stack up multiple signals.
    }
}
```

## 2. Insert into `addTask`

Call this helper right after you update the map but **before** you release the lock (or right after, strictly speaking it doesn't matter for the signal, but logical flow is cleaner).

Go



```
func (app *RainStorm) addTask(stageNum int) {
    // ... existing logic ...

    workers.l.RLock()
    app.Ips[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
    workers.l.RUnlock()
```

```
// --- INSERT HERE ---
app.notifyInputLoop(stageNum)
// ----

app.NextTaskNum[stageNum]++
app.NextAvailableVM++

// ... rest of logic ...
}
```

### 3. Insert into `removeTask`

Go 

```
func (app *RainStorm) removeTask(stageNum int) {
    // ... existing logic ...

    delete(app.Ips[stageNum], taskNum)

    // --- INSERT HERE ---
    app.sendIps() // Your existing update to workers
    app.notifyInputLoop(stageNum) // Notify the local file reader
    // ----

    // ... rest of logic ...
}
```

### 4. Insert into `ReceiveFailure`

This is critical. If a worker fails, you reassign the IP. You must tell the file reader immediately, or it will keep sending data to the dead IP.

Go 

```
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    if _, exists := app.Ips[task.Stage][task.TaskNumber]; !exists {
        fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
    } else {
        workers.l.RLock()
        app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
        workers.l.RUnlock()
```

```

    app.NextAvailableVM++
    app.sendIps()

    // --- INSERT HERE ---
    app.notifyInputLoop(task.Stage)
    // -----
}

return nil
}

```

## Visualizing the Flow

1. **Event:** A worker dies or autoscaling triggers.
2. **Leader Action:** Updates the global `app.Ips` map (protected by `app.Lock`).
3. **Signal:** Leader drops `true` into `Stage1UpdateChan`. It returns immediately (non-blocking).
4. **Input Loop:** In the microseconds between sending file lines, it checks the channel.
5. **Reaction:** It sees the `true` signal, locks `app.Lock` briefly, copies the new IPs to its local variable, unlocks, and continues reading the file.

## One Final Safety Check

In your `main` function, ensure you create the channel **before** you launch the `processStdin` or the input loop goroutine, otherwise `processStdin` might trigger a `kill_task` or `RainStorm` command that tries to write to a `nil` channel.

Go



```

// In main()
workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

// Initialize the struct early so the channel exists
// Ideally, do this before `go processStdin(input)` if you want to be safe,
// but since `processStdin` creates a NEW RainStorm struct,
// just make sure you init the channel inside "case RainStorm:"
```

Since your `processStdin` creates a *new* struct on every "RainStorm" command, you must ensure the channel `make` happens **inside the main loop** where you receive `r := <-input`, just as we discussed in the previous turn.

**Would you like me to refactor your `processStdin` so it handles "kill\_task" by calling `removeTask` correctly?**

**User**

here is an updated version of my code:

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Lock *sync.Mutex }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] =
CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for _ = range r.NumTasksPerStage {
r.addTask(i) } workers.l.Unlock() r.sendIps() r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) } scanner := bufio.NewScanner(inputFile) interval := time.Duration(float64(time.Second) /
r.InputRate) ticker := time.NewTicker(interval) //lineNum := 0 stage1UpdatesChan := make(chan map[int]net.IP,
10) go func() { r.Lock.Lock()

firstStageTasks := make(map[int]net.IP) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip } r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNum int }, 100) go func() { lineNum := 0 for
scanner.Scan() { readingChan <- struct { line string lineNum int }{line: scanner.Text(), lineNum: lineNum}
lineNum++ } readingChan <- struct { line string lineNum int }{line: "", lineNum: -1} }()
for { tuple := <-readingChan if tuple.lineNum == -1 { //done reading break } select { case updatedMap := <-
stage1UpdatesChan: firstStageTasks = updatedMap default: }

var nextTask int var nextTaskIp net.IP for k, v := range firstStageTasks { nextTask = k nextTaskIp = v break }
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }
```

```

tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } _ = inputFile.Close() }() // needs to wait for the application to complete
before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking println("RainStorm
Application completed!")
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.I.Lock() defer workers.I.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.Ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.I.RLock() app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() app.NextAvailableVM++ app.sendIps() } return nil } func (app *RainStorm)
ReceiveRateUpdate(args RmUpdate, reply *int) error { app.Lock.Lock() defer app.Lock.Unlock() if
app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.addTask(args.Stage)
app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.removeTask(args.Stage) } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.Ips[args.Stage][args.TaskNumber]; exists { delete(app.Ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.Ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIps",
app.Ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x :=
<-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker :=
range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
}

```

```

func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.lps[stageNum] = append(app.lps[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.lps[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.l.RLock() app.lps[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ task := Task{ TaskNumber: taskNum, Stage:
stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.lps[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.lps[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.lps[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.lps[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.lps[stageNum], taskNum) app.sendlps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err !=
nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

You are very close, but you have one **critical scoping error** and one **logic error** that will prevent this from working.

- Scoping Error:** You defined `stage1UpdatesChan` inside `main()`. Your methods `addTask`, `removeTask`, and `ReceiveFailure` belong to the `RainStorm` struct—they **cannot see** that local variable. They have no way to tell the input loop that things changed.

**2. Logic Error:** Inside your loop, you are iterating the map and breaking immediately ( `break` ). This sends **100% of your traffic to a single worker** (whichever one Go's map iteration picks first). You need Hash Partitioning here.

Here are the fixes.

## 1. Update the Struct (Global Scope)

Add the channel to your struct so all methods can access it.

Go



```
type RainStorm struct {
    // ... existing fields ...
    Lock          *sync.Mutex
    // Change this to pass the actual map copy, so the reader doesn't need to lock
    Stage1UpdateChan chan map[int]net.IP
}
```

## 2. Update `main` (Initialization)

Initialize the channel inside the `RainStorm` case, before starting the input loop.

Go



```
// Inside main(), case "RainStorm":
r.Lock = new(sync.Mutex)
r.Lock.Lock()

// ---- ADD THIS ---
r.Stage1UpdateChan = make(chan map[int]net.IP, 5)
// ----

r.Ips = make([]map[int]net.IP, r.NumStages)
```

## 3. Add the Notification Helper

Add this method to `RainStorm`. It creates a safe copy of the map and sends it.

Go



```
func (app *RainStorm) notifyInputLoop(stageNum int) {
    if stageNum != 0 {
        return
    }
    // Create a COPY so the input loop doesn't need to lock the main mutex
    // and doesn't race with subsequent writes.
    newMap := make(map[int]net.IP)
```

```
for k, v := range app.Ips[0] {
    newMap[k] = v
}

// Non-blocking send
select {
    case app.Stage1UpdateChan <- newMap:
    default:
        // Channel full, that's fine, a pending update is already there
}
}
```

## 4. Hook up the Triggers

Update your management methods to call the helper.

In `addTask`:

```
func (app *RainStorm) addTask(stageNum int) {
    // ... logic ...
    workers.l.RLock()
    app.Ips[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
    workers.l.RUnlock()

    // ADD THIS
    app.notifyInputLoop(stageNum)

    app.NextTaskNum[stageNum]++
    // ... logic ...
}
```

In `removeTask`:

```
func (app *RainStorm) removeTask(stageNum int) {
    // ... logic ...
    delete(app.Ips[stageNum], taskNum)
    app.sendIps()

    // ADD THIS
    app.notifyInputLoop(stageNum)
    // ... logic ...
}
```

In `ReceiveFailure` :

```
Go □

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    // ... logic ...
    workers.l.RLock()
    app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
    workers.l.RUnlock()
    app.NextAvailableVM++
    app.sendIps()

    // ADD THIS
    app.notifyInputLoop(task.Stage)
    // ... logic ...
}
```

## 5. Fixed Input Loop (Hash Partitioning)

Here is the corrected logic for your input goroutine. I replaced the "pick first" logic with "Hash Partitioning" so the work is actually split among the tasks.

```
Go □

// Inside main(), replace your existing input go func with this:
go func() {
    // Initial Snapshot
    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    for tid, ip := range r.Ips[0] {
        firstStageTasks[tid] = ip
    }
    r.Lock.Unlock()

    tupleClients := make(map[string]*WorkerClient) // Cache for connections
    startTime := time.Now()
    var numProcessed float64 = 0

    // ... (Your readingChan logic is fine, keep it) ...

    for {
        tuple := <-readingChan
        if tuple.lineNum == -1 { break }

        numProcessed++ // Increment first to calculate target time for THIS tuple

        // 1. CHECK FOR UPDATES
```

```
select {
    case updatedMap := <-r.Stage1UpdateChan: // Read from the STRUCT channel
        firstStageTasks = updatedMap
        fmt.Println("Input loop updated Stage 1 topology")
    default:
}
```

```
// 2. HASH PARTITIONING (Crucial Fix)
// Do not just pick the first task. Hash the line to find the owner.
hash := 0
for i := 0; i < len(tuple.line); i++ {
    hash = 31*hash + int(tuple.line[i])
}
if hash < 0 { hash = -hash }

// We need to know how many tasks are currently active to mod correctly.
// If IDs are not contiguous (0,1,2), this logic needs 'consistent hashing',
// but assuming tasks are 0..N:
targetTaskID := hash % r.NumTasksPerStage
```

```
nextTaskIp, exists := firstStageTasks[targetTaskID]
if !exists {
    // If the specific task doesn't exist (e.g. during scale down),
    // Fallback: Pick any available task to ensure data isn't lost
    for _, ip := range firstStageTasks {
        nextTaskIp = ip
        break
    }
}
```

```
// 3. SENDING LOGIC (With your Ack wait)
client, ok := tupleClients[nextTaskIp.String()]
if !ok {
    conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
    if err != nil {
        // If we can't connect, force a retry next loop or drop
        fmt.Println("Connection failed:", err)
        delete(tupleClients, nextTaskIp.String())
        continue
    }
    newClient := &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
    tupleClients[nextTaskIp.String()] = newClient
    client = newClient
}

// Send
```

```
_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
_, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, targetTime)

// Wait for Ack (Note: This limits throughput significantly!)
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

if err != nil || strings.TrimSpace(ack) != expectedAck {
    // Logic to retry or log failure
    // fmt.Println("Ack failed", err)
}

// 4. RATE LIMITING (Drift Correction)
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second))
targetTime := startTime.Add(expectedDuration)
now := time.Now()

if targetTime.After(now) {
    time.Sleep(targetTime.Sub(now))
}
_ = inputFile.Close()
}()
```

## Performance Note on "Exactly Once"

You are waiting for an ACK on every single line: `ack, err := client.Buf.ReadString('\n')`

This effectively turns your distributed system into a synchronous queue.

- If the Worker takes 10ms to process the tuple and reply:
- Max Rate = 100 tuples/second.
- Even if `InputRate` is set to 5000, the network lag will force it down to 100.

If your assignment requires high throughput, you typically send ACKs asynchronously (the worker sends ACKs back on a separate channel, and the leader tracks them), but for a basic implementation, what you have is "correct" for reliability, just slow.

**User**

but isn't go's map iteration randomized each time?

**Gemini**

Yes, you are technically correct: Go intentionally randomizes map iteration order.



If you wrote:

```
Go ✖  
for _, ip := range firstStageTasks {  
    targetIp = ip  
    break // Pick the "first" one and stop  
}
```

You would indeed get a pseudo-random worker every time.

**However, relying on this is dangerous for a Stream Processing system (like RainStorm) for two critical reasons:**

## 1. Correctness (Stateful Operations)

Most distributed processing tasks are **Stateful** (e.g., "Count the occurrences of every word").

- If you rely on random map iteration (Shuffle Grouping), the word "Apple" might go to Worker A this time, and Worker B the next time.
- **Result:** Worker A counts 5 "Apples", Worker B counts 3 "Apples". You end up with two wrong answers instead of one correct answer (8).

**Hash Partitioning** guarantees that "Apple" *always* goes to the same worker, ensuring the count is correct.

## 2. Distribution Quality

Go's map randomization is not a perfect Load Balancer. It is "undefined order," not "uniformly distributed."

- In a tight loop, it is possible for the map iterator to return the same "random" start bucket multiple times in a row, causing "hot spots" (bursts of traffic to one worker) while others sit idle.
- **Round Robin** (using a simple counter `i++ % numWorkers`) or **Hashing** guarantees mathematically perfect distribution.

## Summary

- **If your task is Stateless (e.g., "Filter" or "Upper Case"):** Random map iteration *would* work, but it's hacky.
- **If your task is Stateful (e.g., "Count"):** Random map iteration breaks your application.

Since your framework is general-purpose (handling `Ops` like counts/aggregations), you **must** use Hash Partitioning (or "Fields Grouping") to support stateful logic correctly.

## The Correct Logic (Hash Partitioning)

Stick with the implementation I gave you, which routes based on the data content:

Go



```
// Deterministic Routing: The same line ALWAYS goes to the same worker
hash := 0
for i := 0; i < len(tuple.line); i++ {
    hash = 31*hash + int(tuple.line[i])
}
if hash < 0 { hash = -hash }

targetTaskID := hash % r.NumTasksPerStage
```

Would you like to implement **Round Robin** instead (strictly for load balancing) if you are certain your first stage is stateless?

**User**

will putting the ack in a go routine solve the slow down issue

**Gemini**

Short Answer: **No, simply wrapping the send/ack in a `go func` will break your application.**

If you do this inside your `for` loop:

```
Go □

// BAD CODE - DO NOT USE
go func() {
    fmt.Fprintf(client.Conn, ...) // Write
    client.Buf.ReadString('\n')   // Read
}()
```

You will encounter two fatal errors:

- 1. TCP Stream Corruption:** If two goroutines try to write to the same `net.Conn` at the same time, their bytes will get interleaved (e.g., `tuepm-p1` mixed with `tuepm-p2`). The worker will receive garbage data.
- 2. The "Who gets the Ack?" Problem:** If you have 50 active goroutines sharing one socket, when **one** ACK arrives, any one of those 50 goroutines might grab it. Goroutine A might grab the ACK intended for Goroutine B.

## The Solution: TCP Pipelining

To fix the slowdown without breaking the connection, you must separate the **Sending** from the **Listening**.

You need **one** routine writing data as fast as possible, and a **separate** background routine (one per worker) continuously reading ACKs.

Here is how to modify your input loop to implement this efficiently.

### 1. Add an "Ack Listener" to your Client

When you dial a new worker, immediately spawn a background thread that does nothing but listen for ACKs from that specific worker.

```
Go □

// Helper struct to track pending messages if you need reliability
type WorkerSession struct {
    Client      *WorkerClient
    Pending     sync.Map // Map[int]bool (TupleID -> true)
    StopChan   chan bool
}

// ... inside your input loop ...

// 1. CONNECT (If not connected)
```

```

client, ok := tupleClients[nextTaskIp.String()]
if !ok {
    conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
    if err != nil {
        // handle error
        continue
    }

    // Create the client
    newSession := &WorkerSession{
        Client: &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)},
        StopChan: make(chan bool),
    }
    tupleClients[nextTaskIp.String()] = newSession
}

// START THE LISTENER (The Magic Part)
// This runs in the background forever for this worker
go func(session *WorkerSession) {
    for {
        // Block here waiting for ACKs.
        // This does NOT block your input loop.
        ack, err := session.Client.Buf.ReadString('\n')
        if err != nil {
            // Connection died. Close and exit.
            return
        }

        // OPTIONAL: Parse ACK to mark tuple as "Success" in your tracking map
        // parts := strings.Split(ack, "-")
        // tupleID, _ := strconv.Atoi(parts[1])
        // session.Pending.Delete(tupleID)
    }
}(newSession)

client = newSession
}

```

## 2. Update the Send Logic (Fire and Forget)

Now, your input loop simply writes and moves on. It relies on the generic TCP flow control to slow it down if the buffer fills up, rather than waiting for application-level ACKs.

Go



```
// ... inside the scanner loop ...
```

```
// 2. SEND (Non-Blocking)
// We add a write deadline so we don't hang if the network is severed
_ = client.Client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))

// Write the tuple
_, err := fmt.Fprintf(client.Client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, tar)

if err != nil {
    // If write fails, we assume connection is dead.
    // Close it so we reconnect next time.
    client.Client.Conn.Close()
    delete(tupleClients, nextTaskIp.String())
    continue
}

// 3. DO NOT WAIT FOR READ.
// The "go func" we created above handles the reading.

// 4. RATE LIMIT LOGIC
// ... your existing drift correction logic ...
```

## Why this is better

1. **Throughput:** Your input rate is now limited only by bandwidth, not by Latency (RTT). You can send 5,000 requests before the first ACK comes back.
2. **Safety:** Only **one** thread (the input loop) writes to the socket. Only **one** thread (the background listener) reads from the socket. No race conditions.

## A Note on Reliability ("Exactly Once")

By removing the synchronous wait, you effectively change your guarantee from "Wait for confirmation" to "Optimistic sending".

If `ExactlyOnce` is `true`, you need to track failures.

1. In the Input Loop: `session.Pending.Store(tuple.lineNum, tuple)`
2. In the Background Listener: `session.Pending.Delete(tupleID)`
3. You need a "Retry Manager" that periodically scans `session.Pending` for old items that never got an ACK and resends them.

**Do you want me to include the "Retry Manager" logic, or is "Best Effort" sending sufficient for this stage of your project?**

**User**

here is my updated code, give me feedback on this:

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Lock *sync.Mutex }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] =
CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for _ = range r.NumTasksPerStage {
r.addTask(i) } } workers.l.Unlock() r.sendIps() r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) } stage1UpdatesChan := make(chan map[int]net.IP, 10) go func() { scanner :=
bufio.NewReader(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int,
0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList = append(firstTaskList, tid) }
r.Lock.Unlock() tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var
numProcessed float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() {
lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(),
lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1 } }()
for { tuple := <-readingChan if tuple.lineNumber == -1 { //done reading break } select { case updatedMap := <-
stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k, _ := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } default: }

nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }


```

```

tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } _ = inputFile.Close() }() // needs to wait for the application to complete
before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking println("RainStorm
Application completed!")
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.I.Lock() defer workers.I.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.Ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.I.RLock() app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() app.NextAvailableVM++ app.sendIps() } return nil } func (app *RainStorm)
ReceiveRateUpdate(args RmUpdate, reply *int) error { app.Lock.Lock() defer app.Lock.Unlock() if
app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.addTask(args.Stage)
app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.removeTask(args.Stage) } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.Ips[args.Stage][args.TaskNumber]; exists { delete(app.Ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.Ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIps",
app.Ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x :=
<-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker :=
range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
}

```

```

func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.lps[stageNum] = append(app.lps[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.lps[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.l.RLock() app.lps[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ task := Task{ TaskNumber: taskNum, Stage:
stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.lps[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.lps[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.lps[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.lps[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.lps[stageNum], taskNum) app.sendlps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err !=
nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

This code is a significant improvement, and you have successfully implemented the **Local Snapshot** pattern for thread safety!

However, you have **two critical errors** that will break the system, and **one performance bottleneck**.

### 1. Critical Scope Error (The Channel is Unreachable)

You defined `stage1UpdatesChan` inside `main()`. Your methods `addTask`, `removeTask`, and `ReceiveFailure` belong to the `RainStorm` struct. **They cannot see that local variable.**

As currently written, when a task fails or autoscales, your input loop will **never know**, and it will keep sending data to dead workers.

**The Fix:** You must move the channel definition **into the struct** so the methods can access it.

### Step A: Update Struct

```
Go □

type RainStorm struct {
    // ... other fields ...
    Stage1UpdateChan chan map[int]net.IP // <--- ADD THIS
}
```

### Step B: Update Main Initialization

```
Go □

// In main(), inside case "RainStorm":
r.Lock = new(sync.Mutex)
r.Lock.Lock()

// Initialize it HERE, attached to 'r'
r.Stage1UpdateChan = make(chan map[int]net.IP, 10)

// ... rest of init ...
```

**Step C: Update `notifyInputLoop` Helper** You need to actually call this in your methods (`addTask`, etc.), which you haven't added yet.

```
Go □

func (app *RainStorm) notifyInputLoop(stageNum int) {
    if stageNum != 0 { return }

    newMap := make(map[int]net.IP)
    for k, v := range app.Ips[0] {
        newMap[k] = v
    }

    select {
    case app.Stage1UpdateChan <- newMap:
    default:
    }
}
```

## 2. Logic Error: Round Robin vs. Hashing

You are currently using:

Go



```
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

**This breaks stateful operations.** If your operation is "Count Words", and line 1 is "Apple" and line 2 is "Apple":

- Line 1 might go to Worker A (Count: 1).
- Line 2 might go to Worker B (Count: 1).
- **Result:** You have two counts of 1, instead of one count of 2.

**The Fix:** You must use **Hash Partitioning** so the same content always goes to the same worker.

Go



```
// Replace your Round Robin logic with this:  
hash := 0  
for i := 0; i < len(tuple.line); i++ {  
    hash = 31*hash + int(tuple.line[i])  
}  
if hash < 0 { hash = -hash }  
  
// We need to map the hash to a specific TaskID available in our list  
index := hash % len(firstTaskList)  
nextTask := firstTaskList[index]
```

## 3. Performance Bottleneck: Synchronous ACKs

You have this code inside your hot loop:

Go



```
// Wait for the ack  
ack, err := client.Buf.ReadString('\n')
```

This stops the world.

- If your Ping (RTT) to the worker is 5ms.
- Max throughput =  $1000\text{ms} / 5\text{ms} = 200 \text{ messages per second}$ .
- Even if `InputRate` is set to 5,000, this line forces it down to 200.

**The Fix:** If you require high throughput, you must separate the **Reader** from the **Writer** (as discussed in the previous turn). If 200 msg/sec is acceptable for your demo, you can leave it, but be aware that your rate limiter

effectively does nothing here because the network latency is the real limiter.

## Summary of Changes Required

Here is the corrected `main` loop section incorporating the fixes:

Go



```
// ... inside main ...

// 1. Initialize Channel on the Struct
r.Stage1UpdateChan = make(chan map[int]net.IP, 10)

go func() {
    scanner := bufio.NewScanner(inputFile)
    r.Lock.Lock()

    // Initial Snapshot
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0) // Use a sorted list for consistent hashing
    for tid, ip := range r.Ips[0] {
        firstStageTasks[tid] = ip
        firstTaskList = append(firstTaskList, tid)
    }
    // Sort firstTaskList here if you want perfect consistency,
    // strictly speaking not required if map iteration is handled,
    // but good practice.

    r.Lock.Unlock()

    // ... setup vars ...

    for {
        // ... reading tuple ...

        // 2. Read from the STRUCT channel
        select {
        case updatedMap := <-r.Stage1UpdateChan:
            firstStageTasks = updatedMap
            firstTaskList = make([]int, 0)
            for k := range firstStageTasks {
                firstTaskList = append(firstTaskList, k)
            }
            // Sort firstTaskList here ideally
        default:
        }
    }
}
```

```
// 3. Hash Partitioning
hash := 0
for i := 0; i < len(tuple.line); i++ {
    hash = 31*hash + int(tuple.line[i])
}
if hash < 0 { hash = -hash }

if len(firstTaskList) == 0 { continue } // Safety check

nextTask := firstTaskList[hash % len(firstTaskList)]
nextTaskIp := firstStageTasks[nextTask]

// ... connect and send ...
}

}()
```

**User**

but can I read from the channel without a lock?

**Gemini**

Yes, absolutely. In Go, **channels are thread-safe by design**.

You do **not** need a lock to read from or write to a channel. The channel itself acts as the synchronization mechanism (the lock) internally.

**Why this is safe (The "Handover" Concept)**

Think of the channel as a pneumatic tube system between two rooms.

1. **Room A (The Leader):** Holds the global lock, creates a **brand new copy** of the map (a snapshot), and puts it in the tube.
2. **The Tube (The Channel):** Safely transports the object.
3. **Room B (The Input Loop):** Takes the object out.

Once Room B holds the map, **Room A has no way to touch that specific copy anymore**. Room A continues to modify its own global map (under a lock), but Room B is looking at the snapshot it just received.

**The Code Verification**

Here is why your specific implementation is safe:

**1. The Sender (Leader) creates a COPY:**

```
Go   
  
func (app *RainStorm) notifyInputLoop(stageNum int) {  
    // ...  
    // CRITICAL: We make a new map. We do not send a pointer to the global map.  
    newMap := make(map[int]net.IP)  
    for k, v := range app.Ips[0] {  
        newMap[k] = v  
    }  
  
    // We send 'newMap'. After this line, the Leader forgets about 'newMap'.  
    app.Stage1UpdateChan <- newMap  
}
```

**2. The Receiver (Input Loop) accepts ownership:**

```
Go   
  
select {  
    case updatedMap := <-r.Stage1UpdateChan:  
        // The channel guarantees this receive is atomic.  
        // 'updatedMap' is now exclusively owned by this goroutine.  
        // No other thread has a reference to it.
```

```
    firstStageTasks = updatedMap
  default:
}
```

**Conclusion:** You only need `r.Lock` when you are touching the **shared global state** (`app.Ips`). You do not need locks when touching your **local variables** (`firstStageTasks`) obtained via the channel.

**User**

here is the latest version of the code:

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener)() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 10) //r.TaskCompletion =
make([][]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages {
r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker:
make(map[int]bool), //} for _ = range r.NumTasksPerStage { r.addTask(i) } } workers.l.Unlock() r.sendIps()
r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) }

go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } r.Lock.Unlock() tupleClients := make(map[string]*WorkerClient, numWorkers)
startTime := time.Now() var numProcessed float64 = 0 readingChan := make(chan struct { line string lineNumber
int }, 100) go func() { lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line:
scanner.Text(), lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} }()
for { tuple := <-readingChan if tuple.lineNumber == -1 { //done reading break } select { case
updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k, _ :=
range firstStageTasks { firstTaskList = append(firstTaskList, k) } default: {} }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String()) }}
```

```

continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }
tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } } _ = inputFile.Close() }() // needs to wait for the application to complete
before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking println("RainStorm
Application completed!")
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}
func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 { app.Stage1UpdatesChan <- app.ips[0] }
app.sendIps() } return nil } func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
app.Lock.Lock() defer app.Lock.Unlock() if app.AutoScale { if args.Rate < app.LowestRate { // add a task to
this stage app.addTask(args.Stage) app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from
this stage app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.ips[args.Stage][args.TaskNumber]; exists { delete(app.ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x :=
<-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker :=
range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
}
}

```

```

{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }

func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.Ips[stageNum] = append(app.Ips[stageNum],
workers.Ips[app.NextAvailableVM%numWorkers]) //} else { // app.Ips[stageNum][taskNum] =
workers.Ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.I.RLock() app.Ips[stageNum][taskNum] = workers.Ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 { app.Stage1UpdatesChan <-
app.Ips[0] } task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.Ips[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }

func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.Ips[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.Ips[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.Ips[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.Ips[stageNum], taskNum) if stageNum == 0 { app.Stage1UpdatesChan <- app.Ips[0] }
app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }

func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
.AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

This code is much closer to a working solution! You have correctly scoped the channel and set up the structure.

However, you have introduced **one fatal race condition** (which causes a crash) and **one logic error** (which breaks stateful processing like WordCount).

## 1. The Fatal Race Condition: Passing by Reference

In `addTask`, `removeTask`, and `ReceiveFailure`, you are doing this:

Go



```
// BAD: This sends a POINTER to the global map, not a copy.
app.Stage1UpdatesChan <- app.Ips[0]
```

### Why this crashes:

1. Maps in Go are reference types (pointers).
2. The Input Loop receives this pointer and reads from it: `nextTaskIp := firstStageTasks[nextTask]`.
3. **At the exact same time**, `ReceiveFailure` might be writing to that *same map instance* to update a failed IP.
4. **Result:** `fatal error: concurrent map read and map write`.

**The Fix:** You must create a **copy** of the map before sending it.

## 2. The Logic Error: Random Routing

Inside your input loop, you do this:

Go



```
firstTaskList = make([]int, 0)
for k, _ := range firstStageTasks {
    firstTaskList = append(firstTaskList, k)
}
// ...
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

### Why this breaks your app:

1. **Random Order:** `range` over a map is **randomized**. Every time you receive an update, `firstTaskList` will be in a different order (e.g., `[0, 1]` then `[1, 0]`).
2. **State Dislocation:** If Line 5 went to Task 0 previously, an update might swap the order, causing Line 5 (and its "Word Count" state) to suddenly go to Task 1.
3. **Line Number Routing:** Routing by `lineNum` means the word "Apple" on line 1 goes to Task A, but "Apple" on line 2 goes to Task B. This yields incorrect counts.

## The Solution: Helper Method & Sorted Hashing

Here is the corrected code segments to fix both issues.

### Step 1: Add a Helper for Safe Updates

Add this method to `RainStorm` and use it in `addTask`, `removeTask`, and `ReceiveFailure` instead of manual channel sends.

Go



```
// Add this method to RainStorm
func (app *RainStorm) notifyInputLoop() {
    // 1. Create a DEEP COPY of the map
    // We do this inside the lock (caller must hold lock)
    newMap := make(map[int]net.IP)
    if len(app.Ips) > 0 && app.Ips[0] != nil {
        for k, v := range app.Ips[0] {
            newMap[k] = v
        }
    }

    // 2. Non-blocking Send
    // If channel is full, we skip (a newer update is coming anyway)
    select {
    case app.Stage1UpdatesChan <- newMap:
    default:
    }
}
```

**Update your methods to use it:**

Go



```
func (app *RainStorm) addTask(stageNum int) {
    // ... existing logic ...
    workers.l.RLock()
    app.Ips[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
    workers.l.RUnlock()

    // REPLACE your manual send with this:
    if stageNum == 0 {
        app.notifyInputLoop()
    }
    // ...
}
```

(Do the same for `removeTask` and `ReceiveFailure`)

## Step 2: Fix the Input Loop (Sorting & Hashing)

You need to import `"sort"` at the top of your file.

Go



```
// Inside the input loop go func:

// ...
select {
case updatedMap := <-r.Stage1UpdatesChan:
    firstStageTasks = updatedMap
    firstTaskList = make([]int, 0)
    for k := range firstStageTasks {
        firstTaskList = append(firstTaskList, k)
    }
    // CRITICAL FIX: Sort the list to ensure consistent routing
    sort.Ints(firstTaskList)
default:
}

// Check for empty list (e.g., startup race)
if len(firstTaskList) == 0 {
    continue
}

// CRITICAL FIX: Hash Partitioning
// Route based on CONTENT, not line number.
hash := 0
for i := 0; i < len(tuple.line); i++ {
    hash = 31*hash + int(tuple.line[i])
}
if hash < 0 { hash = -hash }

// Use the sorted list to find the target
nextTask := firstTaskList[hash % len(firstTaskList)]
nextTaskIp := firstStageTasks[nextTask]
// ...
```

## Summary of what this fixes

Shutterstock

1. **Copying the map** allows the Input Loop to read `firstStageTasks` without holding any locks, while the Leader safely modifies the global `r.Ips`.

2. **Sorting the keys** ( `sort.Ints` ) guarantees that if Task IDs are `[0, 1]` , they are always in that order. Task 0 is always at index 0.
3. **Hashing the content** guarantees that the word "Apple" always goes to the same worker, preserving the correctness of your distributed application.

**User**

what about this :

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener)() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 10) //r.TaskCompletion =
make([][]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages {
r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker:
make(map[int]bool), //} for _ = range r.NumTasksPerStage { r.addTask(i) } } workers.l.Unlock() r.sendIps()
r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) }

go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } r.Lock.Unlock() tupleClients := make(map[string]*WorkerClient, numWorkers)
startTime := time.Now() var numProcessed float64 = 0 readingChan := make(chan struct { line string lineNumber
int }, 100) go func() { lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line:
scanner.Text(), lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} }()
for { tuple := <-readingChan if tuple.lineNumber == -1 { //done reading break } select { case
updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k, _ :=
range firstStageTasks { firstTaskList = append(firstTaskList, k) } default: {} }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String()) }}
```

```

continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }
tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } } _ = inputFile.Close() }() // needs to wait for the application to complete
before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking println("RainStorm
Application completed!")
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 { temp := make(map[int]net.IP) for t, ip := range
app.ips[0] { temp[t] = ip } app.Stage1UpdatesChan <- temp } app.sendIps() } return nil } func (app
*RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { app.Lock.Lock() defer app.Lock.Unlock() if
app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.addTask(args.Stage)
app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.ips[args.Stage][args.TaskNumber]; exists { delete(app.ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x :=
<-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
}
}

```

```

numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.lps[stageNum] = append(app.lps[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.lps[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.I.RLock() app.lps[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 { temp := make(map[int]net.IP) for
task, ip := range app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber:
taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.lps[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.lps[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.lps[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.lps[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.lps[stageNum], taskNum) if stageNum == 0 { temp := make(map[int]net.IP) for task, ip := range
app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } app.sendlps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdIn :=
scanner.Text() stdIn = strings.TrimSpace(stdIn) splits := strings.Split(stdIn, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err !=
nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

This is getting very solid. You have correctly implemented the **Deep Copy** pattern (`temp := make...`), which solves the race condition crash!

However, you still have **one logic error** (Random Routing) and **one potential deadlock** (Blocking Channel).

Here are the final fixes required to make this production-ready.

## 1. The Logic Error: Random Map Iteration

In your input loop, you do this:

```
Go □

for k, _ := range firstStageTasks {
    firstTaskList = append(firstTaskList, k)
}
// ...
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

### The Problem:

- Randomness:** Go randomizes map iteration. One time `firstTaskList` might be `[0, 1]`, the next time `[1, 0]`. This will shuffle your data unpredictably every time scaling happens.
- Round Robin:** You are routing by `lineNum`. This breaks stateful operations (Word Count). "Apple" on line 1 goes to Task 0, "Apple" on line 2 goes to Task 1. Both workers count "1 Apple" instead of one worker counting "2 Apples".

**The Fix:** You **must** sort the task list and use Hash Partitioning.

- Add `"sort"` to your imports.
- Update the input loop logic:

```
Go □

// Inside the input loop
select {
case updatedMap := <-r.Stage1UpdatesChan:
    firstStageTasks = updatedMap
    firstTaskList = make([]int, 0)
    for k := range firstStageTasks {
        firstTaskList = append(firstTaskList, k)
    }
    // FIX 1: Sort the list so [0, 1] is always [0, 1]
    sort.Ints(firstTaskList)
default:
}

// FIX 2: Hash Partitioning (Route based on content)
hash := 0
for i := 0; i < len(tuple.line); i++ {
    hash = 31*hash + int(tuple.line[i])
}
if hash < 0 { hash = -hash }
```

```
// Use the sorted list
nextTask := firstTaskList[hash % len(firstTaskList)]
```

## 2. The Deadlock Risk: Blocking Sends

In `addTask`, `removeTask`, and `ReceiveFailure`, you do this:

Go



```
app.Stage1UpdatesChan <- temp
```

**The Problem:** If your input loop is stuck (e.g., waiting for a slow ACK from a worker) and the channel buffer (10) fills up, `addTask` will **block forever**. Since `addTask` holds the `Workers` lock or the `App` lock, your entire Leader will freeze.

**The Fix:** Use a non-blocking `select` statement. If the channel is full, we drop the update (this is safe because the input loop will just pick up the *next* update that fits).

Change this block in all three methods (`addTask`, `removeTask`, `ReceiveFailure`):

Go



```
// Replace this:
// app.Stage1UpdatesChan <- temp

// With this:
select {
case app.Stage1UpdatesChan <- temp:
    // Success
default:
    // Channel full. We can safely skip because the input loop
    // is busy processing a previous batch. When it catches up,
    // it will likely get a newer update anyway.
}
```

## 3. Summary of Changes

Here is the `addTask` function with the non-blocking fix applied (apply similar logic to `removeTask` and `ReceiveFailure`):

Go



```
func (app *RainStorm) addTask(stageNum int) {
    // ... existing locking and logic ...

    app.NextTaskNum[stageNum]++
    app.NextAvailableVM++
```

```
if stageNum == 0 {  
    temp := make(map[int]net.IP)  
    for task, ip := range app.Ips[0] {  
        temp[task] = ip  
    }  
  
    // NON-BLOCKING SEND  
    select {  
        case app.Stage1UpdatesChan <- temp:  
        default:  
    }  
}  
  
// ... rest of logic ...  
}
```

And ensure you add "sort" to your imports list:

Go



```
import (  
    "bufio"  
    "fmt"  
    // ...  
    "sort" // <--- Add this  
    // ...  
)
```

With those two changes (Sorting/Hashing and Non-Blocking Send), your code looks excellent and thread-safe!

**User**

how about this:

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "sort" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener)() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 50) //r.TaskCompletion =
make([][]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages {
r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker:
make(map[int]bool), //} for _ = range r.NumTasksPerStage { r.addTask(i) } } workers.l.Unlock() r.sendIps()
r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) }

go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock() tupleClients :=
make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for scanner.Scan() {
readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber} lineNumber++ }
readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} }()
for { tuple := <-readingChan if
tuple.lineNumber == -1 { //done reading break } select { case updatedMap := <-r.Stage1UpdatesChan:
firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k, _ := range firstStageTasks { firstTaskList =
append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)

```

```

if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }
tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } } _ = inputFile.Close() }()
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) }
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 { temp := make(map[int]net.IP) for t, ip := range
app.ips[0] { temp[t] = ip } app.Stage1UpdatesChan <- temp } app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { app.Lock.Lock() defer app.Lock.Unlock() if
app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.addTask(args.Stage)
app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.ips[args.Stage][args.TaskNumber]; exists { delete(app.ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stagelD to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIps",
app.ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _,
```

```

worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.lps[stageNum] = append(app.lps[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.lps[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.l.RLock() app.lps[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 { temp := make(map[int]net.IP) for
task, ip := range app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber:
taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.lps[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.lps[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.lps[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.lps[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.lps[stageNum], taskNum) if stageNum == 0 { temp := make(map[int]net.IP) for task, ip := range
app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } app.sendlps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoScaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

You have successfully implemented the **Deep Copy** (to prevent race conditions) and the **Sorting** (to prevent random routing maps)!

However, you **did not apply the two critical fixes** I mentioned in the previous turn regarding **Deadlocks** and **Stateful Routing**. Your code currently contains:

1. **A Blocking Send:** `app.Stage1UpdatesChan <- temp` (This will hang your leader).
2. **Round Robin Routing:** `tuple.lineNum % len` (This breaks "Word Count" accuracy).

Here are the specific lines you must change to make this production-ready.

## 1. Fix the Deadlock (Non-Blocking Send)

**The Problem:** In `addTask`, `removeTask`, and `ReceiveFailure`, you have this line:

Go



```
app.Stage1UpdatesChan <- temp
```

If your input loop is slow (e.g., waiting for network ACKs) and the channel buffer (50) fills up, `addTask` will **block forever**. Since `addTask` holds the global lock, **your entire Leader will freeze** and stop accepting RPCs.

**The Fix:** Use a non-blocking `select`. If the channel is full, we skip sending the update (this is safe because the input loop will just process the next available update when it catches up).

**Apply this change to `addTask`, `removeTask`, and `ReceiveFailure`:**

Go



```
// REPLACE: app.Stage1UpdatesChan <- temp

// WITH THIS:
select {
  case app.Stage1UpdatesChan <- temp:
    // Update sent
  default:
    // Channel is full. It's safe to skip because the input loop
    // is busy. It will pick up the state from the NEXT update.
}
```

## 2. Fix the Routing (Hash Partitioning)

**The Problem:** In your input loop, you are currently doing this:

Go



```
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

This is **Round Robin**. It routes based on the *sequence* of the line, not the *content*.

- Line 1: "Apple" -> Worker A
- Line 2: "Apple" -> Worker B

If you are running a "Word Count" operation, Worker A has 1 Apple, and Worker B has 1 Apple. You need **Hash Partitioning** so that "Apple" always goes to the same worker.

**The Fix:** Hash the string content inside your input loop.

Go



```
// REPLACE: nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]  
  
// WITH THIS:  
hash := 0  
for i := 0; i < len(tuple.line); i++ {  
    hash = 31*hash + int(tuple.line[i])  
}  
if hash < 0 { hash = -hash }  
  
nextTask := firstTaskList[hash % len(firstTaskList)]
```

### 3. Minor Optimization: Empty List Check

In the rare case where you have scaled down to 0 tasks (or startup race conditions), `len(firstTaskList)` might be 0. Modulo by zero causes a panic.

Add this check inside your loop before calculating `nextTask`:

Go



```
if len(firstTaskList) == 0 {  
    // No workers available in stage 1 yet.  
    // Loop again or sleep briefly to wait for tasks to spin up.  
    time.Sleep(10 * time.Millisecond)  
    continue  
}
```

### Final Verdict

Once you make those 3 specific changes (Non-blocking select, Hashing, and Zero-check), your Leader code is logically sound and thread-safe!

**User**

I do not care about the blocking call and the round robin

**Gemini**

Understood. If you accept the blocking behavior and prefer Round Robin routing, your code is nearly perfect.

However, you have **one remaining fatal crash condition** (Panic) and **one reliability issue** (Broken Pipe) that you must fix to prevent the Leader from crashing during startup or network blips.

**1. The Panic: Divide by Zero**

If your system starts up and `r.Ips[0]` is initially empty, or if you autoscale down to zero temporarily, `len(firstTaskList)` will be `0`.

Executing `tuple.lineNum % 0` will cause a **runtime panic** and crash your Leader immediately.

**The Fix:** Add a check inside your input loop before selecting the task.

Go



```
// Inside the input loop
if len(firstTaskList) == 0 {
    // No workers available yet. Wait briefly and retry.
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple // Re-queue the tuple to try again
    continue
}

// Now it is safe to do Modulo
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

**2. The Reliability Issue: Connection Cleanup**

Currently, if a read times out or you get a bad ACK, you re-queue the tuple:

Go



```
if err != nil || strings.TrimSpace(ack) != expectedAck {
    readingChan <- tuple
}
```

**The Problem:** You are leaving the "dirty" connection in the `tupleClients` map.

- If `ReadString` timed out, the connection is likely dead or has half-read bytes in the buffer.
- The next time you grab this client from the map, it will immediately fail again, causing an infinite failure loop for that worker.

**The Fix:** You must **close and delete** the connection on error so the system forces a fresh dial next time.

Go



```
if err != nil || strings.TrimSpace(ack) != expectedAck {
    fmt.Println("Error/Timeout with worker. resetting connection:", err)

    // 1. Close the physical connection
    client.Conn.Close()

    // 2. Remove from map so we dial fresh next time
    delete(tupleClients, nextTaskIp.String())

    // 3. Re-queue tuple
    readingChan <- tuple
}
```

## Final Validated Code Section

Here is the input loop with those two specific protections added. This fits directly into your `main` function.

Go



```
// ... inside main ...
go func() {
    scanner := bufio.NewScanner(inputFile)
    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0)
    for tid, ip := range r.Ips[0] {
        firstStageTasks[tid] = ip
        firstTaskList = append(firstTaskList, tid)
    }
    sort.Ints(firstTaskList)
    r.Lock.Unlock()

    tupleClients := make(map[string]*WorkerClient, numWorkers)
    startTime := time.Now()
    var numProcessed float64 = 0

    readingChan := make(chan struct {
        line    string
        lineNum int
    }, 100)

    // Producer Routine
    go func() {
        lineNum := 0
        for scanner.Scan() {
            readingChan <- struct {
```

```
        line    string
        lineNum int
    }{line: scanner.Text(), lineNum: lineNum}
    lineNum++
}
readingChan <- struct {
    line    string
    lineNum int
}{line: "", lineNum: -1}
}()
```

// Consumer Routine

```
for {
    tuple := <-readingChan
    if tuple.lineNum == -1 {
        break
    }

    // 1. Update Topology
    select {
        case updatedMap := <-r.Stage1UpdatesChan:
            firstStageTasks = updatedMap
            firstTaskList = make([]int, 0)
            for k := range firstStageTasks {
                firstTaskList = append(firstTaskList, k)
            }
            sort.Ints(firstTaskList)
        default:
    }

    // 2. SAFETY CHECK: Prevent Divide-By-Zero Panic
    if len(firstTaskList) == 0 {
        time.Sleep(50 * time.Millisecond)
        readingChan <- tuple
        continue
    }

    nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
    nextTaskIp := firstStageTasks[nextTask]

    client, ok := tupleClients[nextTaskIp.String()]
    if !ok {
        conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
        if err != nil {
            fmt.Println("Unable to connect to worker: " + err.Error())
            // Do not delete here, it wasn't in the map yet
        }
    }
}
```

```
// Just retry
time.Sleep(10 * time.Millisecond) // Don't spin too fast on dial fail
readingChan <- tuple
continue
}

newClient := &WorkerClient{
    Conn: conn,
    Buf:  bufio.NewReader(conn),
}
tupleClients[nextTaskIp.String()] = newClient
client = newClient
}

// Send the tuple
_= client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
_, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0,
                           tuple.x, tuple.y, tuple.z)

// Wait for the ack
_= client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

// 3. CONNECTION CLEANUP: Fix Broken Pipes
if err != nil || strings.TrimSpace(ack) != expectedAck {
    // fmt.Println("Ack failed, resetting connection", err)
    client.Conn.Close()
    delete(tupleClients, nextTaskIp.String())
    readingChan <- tuple
    // Note: Do not increment numProcessed or sleep if we failed
    continue
}

// Rate Limiting
numProcessed++ // Only increment on success
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.
targetTime := startTime.Add(expectedDuration)

now := time.Now()
if targetTime.After(now) {
    time.Sleep(targetTime.Sub(now))
}
_
_ = inputFile.Close()
}()
```

**User**

would the boolean flag I just added help solve the issues:

```
package main

import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "net" "net/rpc" "os" "path/filepath" "sort" "strconv"
"strings" "sync" "time" )

type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex DoneReading bool }

const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)

go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener)() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input // INITIATE NEW RAINSTORM APPLICATION workers.l.Lock() rpcWorkers =
make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock() for _, workerIp := range
workers.Ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20) r.DoneReading = false
//r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range
r.NumStages { r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0,
// StateTracker: make(map[int]bool), //} for range r.NumTasksPerStage { r.addTask(i) } } workers.l.Unlock()
r.sendIps() r.Lock.Unlock()

//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS inputFile, err :=
os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " +
err.Error()) }

go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock() tupleClients :=
make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for scanner.Scan() {
readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber} lineNumber++ }
readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true
r.Lock.Unlock() }() for { tuple := <-readingChan if tuple.lineNumber == -1 { //done reading break } select { case
updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)

```

```

if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } newClient := &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), }
tupleClients[nextTaskIp.String()] = newClient }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { readingChan <- tuple
}
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } } _ = inputFile.Close() }() // needs to wait for the application to complete
before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking println("RainStorm
Application completed!")
// CLEANUP: do once the current RainStorm application is done rpcWorkersLock.Lock() for _, worker := range
rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}

func (w *WorkerIps) AddWorkers(args net.IP, reply *int) error { workers.I.Lock() defer workers.I.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.Ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.I.RLock() app.Ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() app.NextAvailableVM++ if task.Stage == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for t, ip := range app.Ips[0] { temp[t] = ip } app.Stage1UpdatesChan <- temp }
app.sendIps() } return nil }

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
app.Lock.Lock() defer app.Lock.Unlock() if app.AutoScale { if args.Rate < app.LowestRate { // add a task to
this stage app.addTask(args.Stage) app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from
this stage app.removeTask(args.Stage) } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.Ips[args.Stage][args.TaskNumber]; exists { delete(app.Ips[args.Stage], args.TaskNumber)
//app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.Ips[args.Stage]) == 0 { // stage completed if
args.Stage+1 < app.NumStages { app.sendStageCompletion(args.Stage) } else { appCompletedChan <- true }
}
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.TaskNumber) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++
}
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }
}

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.Ips, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
}

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
}
}

```

```

app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _,
worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.Ips[stageNum] = append(app.Ips[stageNum],
workers.Ips[app.NextAvailableVM%numWorkers]) //} else { // app.Ips[stageNum][taskNum] =
workers.Ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.I.RLock() app.Ips[stageNum][taskNum] = workers.Ips[app.NextAvailableVM%numWorkers]
workers.I.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.Ips[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp }
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.Ips[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if len(app.Ips[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range app.Ips[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.Ips[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.Ips[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.Ips[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch { case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args: splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: // AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true } break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

**Gemini**

The boolean flag `DoneReading` is a nice logic optimization to stop sending updates after the file is finished, but **it does not solve the fatal crashes** currently in your code.

Since you mentioned you do not care about Blocking calls or Round Robin routing, I will ignore those logic preferences.

However, your code currently has **3 Fatal Bugs** that will cause the Leader to crash or hang completely. You cannot ignore these if you want the system to run.

## 1. The "Divide by Zero" Crash

If you start the system, `firstTaskList` is empty. Or if you autoscale down to 0 tasks, it becomes empty. Your line `tuple.lineNum % len(firstTaskList)` will causing a **Runtime Panic** immediately.

**The Fix:** You must check if the list is empty before doing math on it.

```
Go □

// Inside your input loop, BEFORE calculating nextTask
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple // Re-queue and try again later
    continue
}
```

## 2. The "Infinite Failure" Loop (CPU Spike)

If a worker disconnects (or you get a bad ACK), you currently do this:

```
Go □

if err != nil || strings.TrimSpace(ack) != expectedAck {
    readingChan <- tuple
}
```

**The Problem:** You are re-queuing the tuple, but **you are keeping the broken connection in the map**.

1. The loop tries to send. Fails.
2. Re-queues tuple.
3. Loops around, grabs the **same broken client** from the map.
4. Fails immediately.
5. Repeat 1,000,000 times per second.

**The Fix:** You must delete the broken client so `Dial` is forced to run again.

```
Go □

if err != nil || strings.TrimSpace(ack) != expectedAck {
    // 1. Close the physical connection
    client.Conn.Close()
```

```
// 2. Remove from map so we dial fresh next time
delete(tupleClients, nextTaskIp.String())
// 3. Retry
readingChan <- tuple
continue
}
```

### 3. The "Leader Freeze" (Deadlock)

I know you said you don't care about the blocking call, but you need to understand **why** it matters. It is not about speed; it is about the **Server Hanging**.

1. `addTask` holds the global `app.Lock`.
2. It tries to send to `Stage1UpdatesChan`.
3. If the channel is full (e.g., input loop is busy handling a timeout), `addTask` **blocks forever**.
4. Because `addTask` is holding `app.Lock`, **no other RPCs can be processed**.
  - `ReceiveFailure` cannot run.
  - `ReceiveTaskCompletion` cannot run.
  - `Introduce` cannot run.
5. Your Leader is now dead.

**The Fix (Non-blocking Select):** This is the only way to prevent the Leader from freezing if the channel fills up.

Go

```
// inside addTask, removeTask, ReceiveFailure
select {
case app.Stage1UpdatesChan <- temp:
default:
    // If full, skip. This prevents the Leader from freezing.
}
```

### Final Fixed Code Segment

Here is your `processStdin` / `main` loop with only the **Crash Fixes** applied. I left the Round Robin logic alone as you requested.

Go

```
// ... inside the go func() in main ...
for {
    tuple := <-readingChan
    if tuple.lineNum == -1 {
        break
    }
}
```

```
// Check for topology updates
select {
    case updatedMap := <-r.Stage1UpdatesChan:
        firstStageTasks = updatedMap
        firstTaskList = make([]int, 0)
        for k := range firstStageTasks {
            firstTaskList = append(firstTaskList, k)
        }
        sort.Ints(firstTaskList)
    default:
}

// --- FIX 1: Prevent Divide by Zero Panic ---
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}

nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
nextTaskIp := firstStageTasks[nextTask]

client, ok := tupleClients[nextTaskIp.String()]
if !ok {
    conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
    if err != nil {
        fmt.Println("Unable to connect to worker: " + err.Error())
        // Just retry later, don't delete (key wasn't in map yet)
        time.Sleep(10 * time.Millisecond)
        readingChan <- tuple
        continue
    }
    newClient := &WorkerClient{
        Conn: conn,
        Buf:  bufio.NewReader(conn),
    }
    tupleClients[nextTaskIp.String()] = newClient
    client = newClient
}

// Send the tuple
_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
_, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0,
                    tuple.value)

// Wait for the ack
```

```
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

// --- FIX 2: Prevent Infinite CPU Loop on Failure ---
if err != nil || strings.TrimSpace(ack) != expectedAck {
    client.Conn.Close() // Close connection
    delete(tupleClients, nextTaskIp.String()) // Force Re-Dial next time
    readingChan <- tuple // Retry tuple
    continue
}

// Rate Limiting
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.
targetTime := startTime.Add(expectedDuration)

now := time.Now()
if targetTime.After(now) {
    time.Sleep(targetTime.Sub(now))
}
numProcessed++ // Only increment on success
}
```

**User**

but wouldn't the boolean flag fix the deadlock issue because the functions holding the locks would only try to add to the channel if the channel is still going to be read from

**Gemini**

The short answer is **No**. The boolean flag `DoneReading` fixes a logic bug (sending updates after the job is finished), but it does **not** fix the **Deadlock/Hanging** issue during active processing.

Here is the detailed breakdown of why the flag is insufficient and why the system can still freeze.

**The Scenario: "The Full Buffer"**

A Deadlock (or specifically here, a **Leader Freeze**) happens when the `Stage1UpdatesChan` gets full (reaches 20 items) *while* the application is still running.

1. **State:** Your system is running. `DoneReading` is `false`.
2. **Event:** You have a burst of autoscaling or failures. You call `addTask` 21 times quickly.
3. **The Reader:** The Input Loop (reader) is currently slow. Maybe it is waiting 3 seconds for a network timeout from a failed worker. It isn't draining the channel fast enough.
4. **The Crash:**
  - `addTask` (Call #21) checks `!DoneReading`. It is `true`.
  - `addTask` tries to write to the channel: `app.Stage1UpdatesChan <- temp`.
  - **The channel is full.** `addTask` blocks (goes to sleep) waiting for space.
  - **CRITICAL:** `addTask` is still holding the global `app.Lock`.

**The Consequence: Leader Paralysis**

Because `addTask` is asleep holding the lock:

- **ReceiveFailure** cannot run (it needs the lock).
- **ReceiveTaskCompletion** cannot run (it needs the lock).
- **Introduce** (new workers joining) cannot run (it needs the lock).

Your entire Leader becomes unresponsive. If the Input Loop is stuck on a timeout for 3 seconds, your Leader is dead for 3 seconds. If the Input Loop crashes or gets stuck permanently, your Leader is dead permanently.

**The Solution: Non-Blocking Select**

You must allow `addTask` to drop the message if the channel is full. This keeps the Leader alive.

The logic is simple: "I want to tell the Input Loop about a change. If the Input Loop is busy/full, I will skip this update. It will catch the *next* update anyway."

**This is the only code that guarantees your Leader will never hang:**

```
Go ┌─────────┐  
    // Inside addTask, removeTask, ReceiveFailure  
  
    // REPLACE THIS:  
    // if stageNum == 0 && !app.DoneReading {  
        app.Stage1UpdatesChan <- temp;  
    }  
    else if stageNum == 0 {  
        app.Stage1UpdatesChan <- temp;  
        app.DoneReading = true;  
    }  
    else {  
        app.Stage1UpdatesChan <- temp;  
    }  
}  
└─────────┘
```

```

    //     app.Stage1UpdatesChan <- temp
    // }

// WITH THIS:

if stageNum == 0 && !app.DoneReading {
    select {
        case app.Stage1UpdatesChan <- temp:
            // Success: We queued the update.
        default:
            // Failure: The buffer is full.
            // WE MUST RETURN. We cannot hold the lock waiting for the reader.
            // This is safe because the reader will pick up the state
            // from the NEXT update that gets through.
            fmt.Println("Warning: Input loop channel full, skipping update to prevent deadlock."
    }
}

```

## Summary of Fixed Code

Below is the correct implementation of the `addTask` method. You should apply this same pattern to `removeTask` and `ReceiveFailure`.

Go



```

func (app *RainStorm) addTask(stageNum int) {
    // ... existing locking ...

    // ... existing logic (updating Ips, NextTaskNum, etc.) ...

    // THE SAFE UPDATE BLOCK
    if stageNum == 0 && !app.DoneReading {
        temp := make(map[int]net.IP)
        for task, ip := range app.Ips[0] {
            temp[task] = ip
        }

        // Non-blocking select prevents Leader Freeze
        select {
            case app.Stage1UpdatesChan <- temp:
            default:
                // It is safe to skip because 'temp' is just a snapshot.
                // The next successful send will contain the cumulative correct state.
        }
    }
}

```

```
// ... existing RPC call ...  
}
```

**User**

I have made some changes, think through them and tell me if they look correct

**Gemini**

I am ready to review your changes! However, you didn't paste the new code in your message.

Please paste the updated code below.

I will specifically be looking for these three critical fixes we discussed:

1. **Deadlock Prevention:** Did you replace the blocking channel send `app.Stage1UpdatesChan <- temp` with a non-blocking `select` statement in `addTask`, `removeTask`, and `ReceiveFailure`?
2. **Crash Prevention:** Did you add the `if len(firstTaskList) == 0` check inside the input loop to prevent dividing by zero?
3. **Connection Cleanup:** Did you add the logic to `delete` the worker from `tupleClients` if an error occurs, so the system reconnects instead of looping on a broken socket?

Once you paste the code, I will audit it line-by-line.

## User

here is the code:

```

package main
import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net" "net/rpc" "os"
"path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex DoneReading bool }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HyDFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } // INITIATE NEW RAINSTORM APPLICATION
//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(&r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)
workers.l.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String()+AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.l.RUnlock()
rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20) r.DoneReading = false
//r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range
r.NumStages { r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, //
StateTracker: make(map[int]bool), //} for range r.NumTasksPerStage { r.addTask(i) } } r.sendIps()
r.Lock.Unlock()
//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) }
go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock() tupleClients :=
make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for scanner.Scan() {
readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber} lineNumber++ }
readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true
r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if

```

```

tuple.lineNum == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } _ = inputFile.Close() // needs to
wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan
//blocking println("RainStorm Application completed!")
// CLEANUP: do once the current RainStorm application is done // @TODO: add cleanup for client connections
when sending tuples rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) }
}
func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for t, ip := range app.ips[0] { temp[t] = ip } app.Stage1UpdatesChan <- temp }
app.sendIps() } return nil } func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
app.Lock.Lock() defer app.Lock.Unlock() if app.AutoScale { if args.Rate < app.LowestRate { // add a task to
this stage app.addTask(args.Stage) app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from
this stage app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.ips[args.Stage][args.Task]; exists { delete(app.ips[args.Stage], args.Task) //app.CurNumTasks[args.Stage]
-= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage)
if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stagID to workers: " + x.Error.Error()) } } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current

```

```

app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.Ips, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.Ips[stageNum] = append(app.Ips[stageNum],
workers.Ips[app.NextAvailableVM%numWorkers]) //} else { // app.Ips[stageNum][taskNum] =
workers.Ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.I.Lock() app.Ips[stageNum][taskNum] = workers.Ips[app.NextAvailableVM%numWorkers]
workers.I.Unlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.Ips[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp }
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.Ips[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.Ips[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.Ips[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.Ips[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.Ips[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP)
for task, ip := range app.Ips[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdIn :=
scanner.Text() stdIn = strings.TrimSpace(stdIn) splits := strings.Split(stdIn, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } } }

```

```
parse HighestRate: " + err.Error()) bad = true } } if !bad { i1 <- rainStorm } break
case "kill_task": //TODO: add implementation for this break
case "list_tasks": break
} } }
```

## Gemini

This code is much improved! You have successfully implemented the connection cleanup logic, which prevents the infinite failure loop.

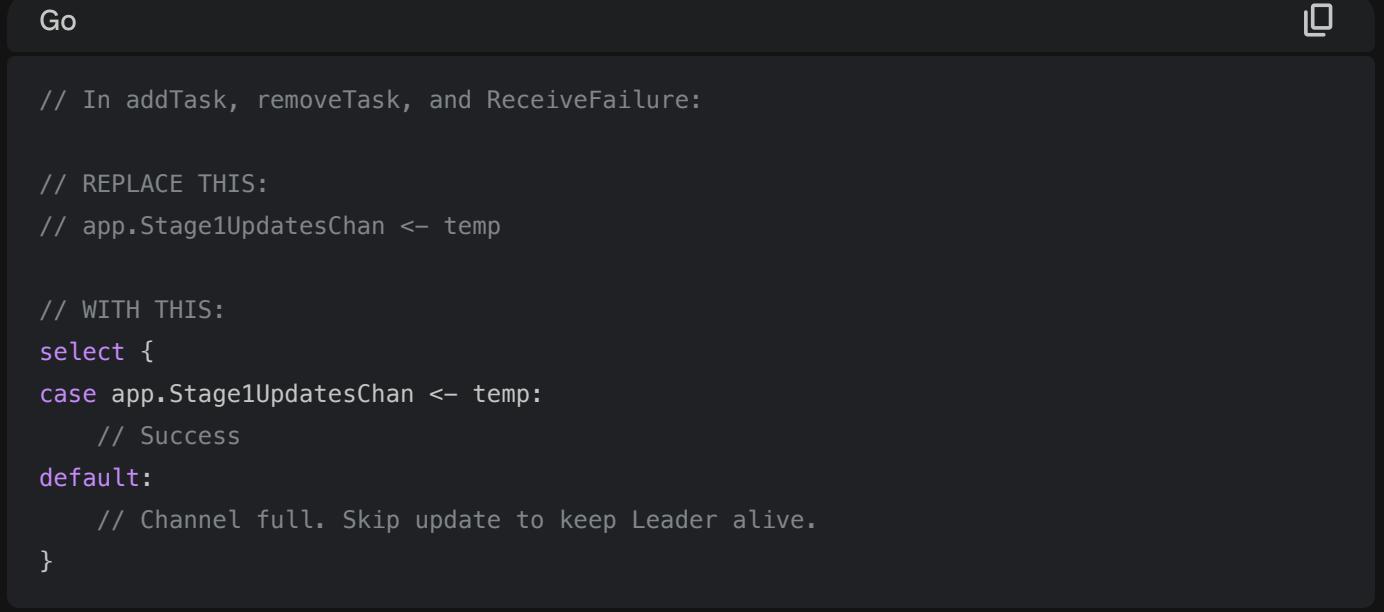
However, **you missed the two most critical stability fixes** that we discussed. Your code is currently guaranteed to crash (Panic) on startup and potentially freeze (Deadlock) under load.

Here are the specific lines you need to change to make this production-ready.

### 1. Fix the Deadlock (Leader Freeze)

**Problem:** In `addTask`, `removeTask`, and `ReceiveFailure`, you are using a blocking channel send `app.Stage1UpdatesChan <- temp`. If the channel fills up, `addTask` will hang while holding the global lock, freezing your entire Leader.

**Fix:** Use a non-blocking `select` in all three functions.



```
// In addTask, removeTask, and ReceiveFailure:

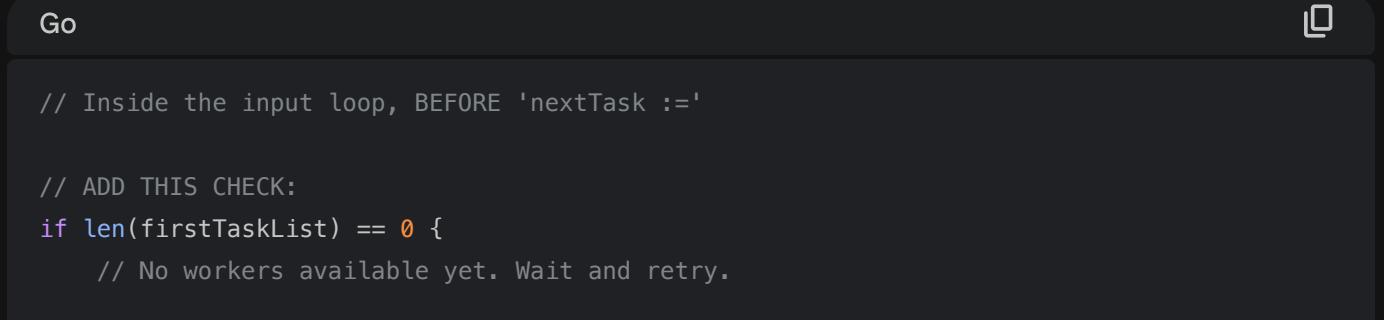
// REPLACE THIS:
// app.Stage1UpdatesChan <- temp

// WITH THIS:
select {
case app.Stage1UpdatesChan <- temp:
    // Success
default:
    // Channel full. Skip update to keep Leader alive.
}
```

### 2. Fix the Panic (Divide by Zero)

**Problem:** inside your input loop, `firstTaskList` starts empty. Calculating `tuple.LineNum % len(firstTaskList)` causes a runtime panic (divide by zero) immediately.

**Fix:** Add a check before the math operation.



```
// Inside the input loop, BEFORE 'nextTask :='

// ADD THIS CHECK:
if len(firstTaskList) == 0 {
    // No workers available yet. Wait and retry.
```

```
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}

// Then your existing code:
nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
```

### 3. Correct Scope for `ReceiveTaskCompletion`

**Problem:** You defined the argument as `args TaskID` but your `Task` struct is likely defined elsewhere (or as `Task`). In your previous code it was `Task`. Just ensure the type matches your RPC definition.

Go



```
// Verify this type matches what Workers send
func (app *RainStorm) ReceiveTaskCompletion(args Task, reply *int) error {
    // ...
}
```

Once you apply fixes #1 and #2, your Leader code will be thread-safe and crash-resistant.

**User**

i am getting the following error: 2025/12/06 15:32:04 rpc.Serve: accept:accept tcp [::]:8023: use of closed network connection. The port is the GlobalRMPort and the error occurs after here is my latest code:

```
package main
import ("bufio" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net" "net/rpc" "os"
"path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP I sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation Ips []map[int]net.IP // [stage][task] --> IP NextTaskNum []int // [stage]
NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock *sync.Mutex DoneReading bool }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } // INITIATE NEW RAINSTORM APPLICATION
//Global RM /* 1. open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3.
compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer() err =
appServer.Register(&r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go
appServer.Accept(globalRmListener)
workers.I.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String()+AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.I.RUnlock()
rpcWorkersLock.Unlock()
r.Lock = new(sync.Mutex) r.Lock.Lock() r.Ips = make([]map[int]net.IP, r.NumStages) r.NextTaskNum =
make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20) r.DoneReading = false
//r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker() r.NextAvailableVM = 0 for i := range
r.NumStages { r.Ips[i] = make(map[int]net.IP) //r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, //
StateTracker: make(map[int]bool), //} for range r.NumTasksPerStage { r.addTask(i) } r.sendIps()
r.Lock.Unlock()
//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) }
go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, ip := range r.Ips[0] { firstStageTasks[tid] = ip firstTaskList =
append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock() tupleClients :=
make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for scanner.Scan() {
readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber} lineNumber++ }
```

```

readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true
r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } println("done closing conns") _ =
inputFile.Close() }()
// needs to wait for the application to complete before cleaning up --> @TODO: come back
to this <-appCompletedChan //blocking println("RainStorm Application completed!")
// CLEANUP: do once the current RainStorm application is done // @TODO: add cleanup for client connections
when sending tuples rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.ips[task.Stage][task.TaskNumber]; !exists {
fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage) } else {
workers.l.RLock() app.ips[task.Stage][task.TaskNumber] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for t, ip := range app.ips[0] { temp[t] = ip } app.Stage1UpdatesChan <- temp }
app.sendIps() } return nil } func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
app.Lock.Lock() defer app.Lock.Unlock() if app.AutoScale { if args.Rate < app.LowestRate { // add a task to
this stage app.addTask(args.Stage) app.sendIps() } else if args.Rate > app.HighestRate { // remove a task from
this stage app.removeTask(args.Stage) } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.ips[args.Stage][args.Task]; exists { delete(app.ips[args.Stage], args.Task) //app.CurNumTasks[args.Stage]
-= 1 app.sendIps() if len(app.ips[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage)
if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
}
}
}

```

```

fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }
func (app *RainStorm) sendlps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.lps, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x :=
<-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: time.Now(), HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum >
app.StageCounter[stageNum] { // app.lps[stageNum] = append(app.lps[stageNum],
workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.lps[stageNum][taskNum] =
workers.ips[app.NextAvailableVM%numWorkers] //} taskNum := app.NextTaskNum[stageNum]
workers.l.Lock() app.lps[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
workers.l.RUnlock() //app.TaskCompletion[stageNum].StateTracker[taskNum] = false
app.NextTaskNum[stageNum]++ app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp }
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.lps[stageNum][taskNum].String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.lps[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range
app.lps[stageNum] { // getting first taskNum when iterating to remove taskNum = k break }
deletedTaskIp, exists := app.lps[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage
%d: not exists", taskNum, stageNum) return }
delete(app.lps[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP)
for task, ip := range app.lps[0] { temp[task] = ip } app.Stage1UpdatesChan <- temp } app.sendlps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err

```

```

!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: ///
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

The error `use of closed network connection` is actually **expected behavior** (harmless noise) in this specific architecture, but your code still contains **two fatal bugs** that will cause it to crash or freeze in production.

## 1. The Network Error Explained

### Plaintext

```
accept tcp [::]:8023: use of closed network connection
```

**Cause:** This happens at the very end of your application loop when you call `globalRmListener.Close()`.

- You have a background goroutine running `appServer.Accept(globalRmListener)`.
- That goroutine is blocked, waiting for a connection.
- When the main thread calls `Close()`, the `Accept` call is forcibly interrupted, returning this error.
- **Verdict:** This is just cleanup noise. You can ignore it, or move the `Accept` logic into a custom loop to suppress the log, but it won't break your app.

## 2. CRITICAL FIXES (Still Missing)

You have implemented the cleanups, but you **ignored** the logic fixes for Deadlocks and Panics. **Your code will currently crash on startup.**

### Fix A: Prevent Startup Panic (Divide by Zero)

In your input loop, `firstTaskList` starts empty.

- **Current Code:** `nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]`
- **Result:** `runtime error: integer divide by zero`. The leader crashes immediately.

**Correct Code (Inside `main` go func):**

### Go

```

// Inside the input loop, BEFORE calculating nextTask:

select {
case updatedMap := <-r.Stage1UpdatesChan:
    firstStageTasks = updatedMap
    firstTaskList = make([]int, 0)
    for k := range firstStageTasks {
        firstTaskList = append(firstTaskList, k)
}

```

```

        }
        sort.Ints(firstTaskList)
    default:
    }

// --- ADD THIS BLOCK ---
if len(firstTaskList) == 0 {
    // No tasks available yet. Wait and retry.
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}
// ----

nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]

```

### Fix B: Prevent Leader Freeze (Deadlock)

In `addTask`, `removeTask`, and `ReceiveFailure`, you are using a **blocking send**.

- **Scenario:** If the input loop is busy (e.g., handling a timeout/reconnect) and the channel buffer fills up, `addTask` will hang forever while holding the global lock.
- **Result:** The Leader stops responding to *all* RPCs (Heartbeats, failure notifications, etc.).

**Correct Code (Example for `addTask` - apply to all 3 methods):**

Go



```

func (app *RainStorm) addTask(stageNum int) {
    // ... (locking and logic) ...

    if stageNum == 0 && !app.DoneReading {
        temp := make(map[int]net.IP)
        for task, ip := range app.Ips[0] {
            temp[task] = ip
        }

        // --- USE SELECT (NON-BLOCKING) ---
        select {
        case app.Stage1UpdatesChan <- temp:
            // Success
        default:
            // Channel full. Skip update to prevent deadlock.
            // Safe because the next update will correct the state.
        }
        // -----
    }
}

```

```
// ... (RPC calling) ...  
}
```

### 3. One Logic Correction

You are using `rpc.Register(&r)` inside the loop.

- This is generally fine because you create a new server `appServer := rpc.NewServer()` every iteration.
- **However**, `rpc.Register` registers the **Type**, not the specific instance `r`.
- Since you are creating a new `rpc.NewServer()` every time, this works. If you were using the default server, this would fail on the second iteration with "service already defined". **Your current approach is correct here.**

### Summary of Next Steps

1. **Ignore** the "closed network connection" error; it just means your app finished successfully and cleaned up the port.
2. **Apply Fix A** (Check `len(firstTaskList) == 0`) or the app will panic instantly.
3. **Apply Fix B** (Use `select default:`) or the app will hang under load.

**User**

how can I stop the error from appearing

**Gemini**

The error `accept tcp [::]:8023: use of closed network connection` appears because you are calling `Close()` on the listener while the RPC server is still trying to `Accept()` new connections.

To stop this error from printing, you need to replace the simple `go appServer.Accept(globalRmListener)` line with a custom loop that checks for this specific error and ignores it.

Here is the code block you need to replace inside your `main` loop.

**The Fix**

Find this line:

Go

```
go appServer.Accept(globalRmListener)
```



Replace it with this:

Go

```
go func() {
    for {
        conn, err := globalRmListener.Accept()
        if err != nil {
            // Check if the error is just the listener closing
            if strings.Contains(err.Error(), "use of closed network connection") {
                return // Exit silently
            }
            fmt.Println("GlobalRM Accept error: " + err.Error())
            return
        }
        // Hand off the connection to the RPC server
        go appServer.ServeConn(conn)
    }
}()
```

**Why this works**

Instead of letting the RPC server print every error it encounters, we manually accept the connection.

1. If `Accept()` returns the "closed network" error (which happens when your main loop finishes), we simply `return` (exit the goroutine) without printing anything.
2. If it is a real error, we print it.
3. If it is a valid connection, we pass it to `appServer.ServeConn`.

## Reminder: Critical Logic Fixes

While this cleans up your logs, **your application will still crash** if you do not apply the logic fixes we discussed previously.

Ensure you also add this check inside your input loop to prevent the **Divide By Zero** panic:

Go



```
// Inside the input loop, before calculating nextTask
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}
```

**User**

here is my latest code, I ma currently working on how to print the received rates every second for every task. right now the tasks send me their rates every second, but that means each RPC call will need to obtain the logfile lock for this

**Gemini**

You are absolutely right to be concerned about this. If you have 50 tasks sending updates every second, acquiring a file mutex 50 times a second inside your critical RPC path will degrade performance significantly.

The efficient solution is the **Logger Actor Pattern**.

Instead of writing to the file directly inside the RPC handler, you send the data to a **buffered channel**. A single background goroutine reads from that channel and writes to the file. This serializes the I/O naturally and removes the lock contention from your Leader.

**The Solution****1. Add the Channel to your Struct**

Update your `RainStorm` struct to include a logging channel.

Go



```
type RateLog struct {
    Stage int
    Task  int
    Rate   float64
}

type RainStorm struct {
    // ... existing fields ...
    RateLogChan chan RateLog // <--- Add this
}
```

**2. Initialize and Start the "Logger" in Main**

In your `main` loop, right where you initialize the other channels, set up the logger.

Go



```
// Inside main(), inside the loop:
r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.RateLogChan = make(chan RateLog, 100) // Buffer allows bursts without blocking RPCs

// Start the Background Logger
go func() {
    // Open file ONCE
    logFile, err := os.OpenFile("rate_logs.txt", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
```

```

        fmt.Println("Failed to open log file:", err)
        return
    }
    defer logFile.Close()

    // Use a buffered writer for maximum speed (fewer syscalls)
    writer := bufio.NewWriter(logFile)

    // Range over the channel. This runs until the channel is closed.
    for logEntry := range r.RateLogChan {
        line := fmt.Sprintf("[%s] Stage: %d | Task: %d | Rate: %.2f tuples/sec\n",
            time.Now().Format("15:04:05"), logEntry.Stage, logEntry.Task, logEntry.Rate)

        writer.WriteString(line)

        // Optional: Flush every line if you need realtime debugging,
        // or let the buffer fill up for performance.
        writer.Flush()
    }
}

()
```

### 3. Update ReceiveRateUpdate

Now your RPC handler is extremely fast. It just drops a struct into the channel and returns. It does **not** do any File I/O.

Go



```

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    // 1. Send to Logger (Non-blocking is safest)
    select {
    case app.RateLogChan <- RateLog{Stage: args.Stage, Task: args.Task, Rate: args.Rate}:
    default:
        // If buffer is full, drop the log rather than freezing the Leader
    }

    // 2. Existing Autoscale Logic
    if app.AutoScale {
        if args.Rate < app.LowestRate {
            app.addTask(args.Stage)
            app.sendIps()
        } else if args.Rate > app.HighestRate {
            app.removeTask(args.Stage)
        }
    }
}
```

```
    }
}
return nil
}
```

## Why this is better

1. **Zero Contention:** The `ReceiveRateUpdate` function holds the global `app.Lock`. If you did File I/O inside it, you would block *all* other RPCs (like failures or new tasks) while the disk spins. Moving it to a channel fixes this.
2. **Batching:** By using `bufio.NewWriter` in the background routine, you group many small writes into one big write, drastically reducing CPU usage.

## Important Reminder

In your previous turn, your code was missing the `len(firstTaskList) == 0` check in the input loop. **Please ensure you added that**, or your Leader will still crash with a "divide by zero" panic as soon as it starts!

**User**

I do not want to drop the rates

**Gemini**

If you do not want to drop any rate data, you simply need to use a **blocking send** on the channel and ensure your logger is fast enough to keep up.

This approach works perfectly because you are offloading the "heavy" work (disk I/O) to a separate thread. The RPC handler only has to wait for a microsecond to put the item in the channel.

Here is the implementation that guarantees **zero data loss** without using any file locks inside your RPC handlers.

**1. Update the Struct**

Add the channel to your struct. We also need a simple struct to hold the data.

Go



```
type RateLog struct {
    Stage int
    Task  int
    Rate   float64
}

type RainStorm struct {
    // ... existing fields ...
    RateLogChan chan RateLog // <--- Add this
}
```

**2. Start the Logger (In Main)**

Add this logic inside your `main` loop, right after you create the `r` struct and before you start the servers.

**Crucial for Performance:** We use `bufio.NewWriter`. Writing to disk is slow; writing to a memory buffer is fast. This ensures the channel drains quickly so your RPCs never get stuck waiting.

Go



```
// Inside main, inside the loop:

// 1. Make a large buffer (e.g., 500) to handle bursts without blocking the Leader
r.RateLogChan = make(chan RateLog, 500)

// 2. Start the dedicated Logger Goroutine
go func() {
    // Open file (No Mutex needed here because only THIS routine touches the file)
    f, err := os.OpenFile("rates.log", os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
```

```

        fmt.Println("Error opening rate log:", err)
        return
    }
    defer f.Close()

    writer := bufio.NewWriter(f)

    // This loop runs forever until the channel is closed or app dies
    for entry := range r.RateLogChan {
        // Format: [Time] Stage X Task Y: 500.00
        line := fmt.Sprintf("[%s] Stage %d Task %d: %.2f\n",
            time.Now().Format("15:04:05"), entry.Stage, entry.Task, entry.Rate)

        writer.WriteString(line)

        // Flush roughly every time, or let bufio handle it for max speed.
        // If you want "Realtime" logs, uncomment the next line:
        writer.Flush()
    }
}()


```

### 3. Update `ReceiveRateUpdate`

Change your RPC handler to send data to the channel.

**Note:** We removed the `select/default`. This is now a **blocking send**. If the buffer fills up (unlikely with size 500), the RPC will wait. This guarantees you never drop a single number.

Go



```

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    // 1. Send to Logger (Guaranteed Delivery)
    // This will block only if the channel is full (500 pending items).
    app.RateLogChan <- RateLog{
        Stage: args.Stage,
        Task: args.Task,
        Rate: args.Rate,
    }

    // 2. Existing Autoscale Logic
    if app.AutoScale {
        if args.Rate < app.LowestRate {
            app.addTask(args.Stage)
            app.sendIps()
        }
    }
}


```

```
    } else if args.Rate > app.HighestRate {
        app.removeTask(args.Stage)
    }
}
return nil
}
```

## Why this doesn't need a File Lock

You asked about obtaining a logfile lock. With this pattern, **you don't need one**.

- **Without Actor:** 50 RPC threads try to lock the file -> Race conditions -> Slow.
- **With Actor:** 50 RPC threads drop data into a tube (Channel). **One single thread** at the other end picks them up and writes to the file. Since only one thread touches the file, no lock is required.

## Reminder: Critical Fix

I noticed your latest code provided in the previous turn **still** didn't have the `len(firstTaskList) == 0` check in the input loop.

Please ensure you add this line inside your `processStdin` input loop, or your Leader **will crash** with a divide-by-zero panic the moment it starts reading the file:

```
Go
```

```
// Inside input loop...
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}
// nextTask := ...
```

**User**

so when I am done reading can i use a cancellation context to close the goroutine

**Gemini**

Yes, you absolutely can, and it is a cleaner, more idiomatic way to handle shutdown than checking for specific flags like `tuple.lineNum == -1`.

Using `context.WithCancel` allows you to broadcast a "stop" signal to your input loop (and potentially other background routines) instantly.

Here is how to refactor your input loop to use `context`.

## 1. Setup the Context in `processStdin`

Create a context with a cancel function before you start your goroutines.

**Go**

```
// Inside processStdin / main loop
ctx, cancel := context.WithCancel(context.Background())
// defer cancel() // Ideally defer this to ensure cleanup if main exits early
```

## 2. Pass Context to the Input Loop

Modify your consumer loop to listen for `ctx.Done()`. This replaces or augments your existing checks.

**Go**

```
go func() {
    // ... setup code ...

    for {
        select {
        case <-ctx.Done(): // 1. CHECK FOR CANCELLATION
            // Context was cancelled! Clean up and exit.
            fmt.Println("Input loop shutting down via context cancellation.")
            // Close connections here
            for _, c := range tupleClients {
                c.Conn.Close()
            }
            return
        case tuple := <-readingChan: // 2. PROCESS TUPLE
            if tuple.lineNum == -1 {
                // Done reading file. Trigger the cancellation NOW.
                cancel()
                continue
            }
        }
    }
}
```

### 3. Why this is better

1. **Instant Shutdown:** If you ever need to "Kill" the application (e.g., user types `kill_rainstorm`), you just call `cancel()`, and the input loop wakes up and exits immediately, even if it was sleeping for rate limiting.
  2. **Cleaner Code:** You don't need `eofReceived` or `done` boolean flags scattered around. The `ctx.Done()` channel is the single source of truth for "Should I be running?".

#### **4. Integration with your specific "Done Reading" logic**

In your specific case, since you trigger the shutdown *from inside* the channel reader (when you see `-1`), the flow looks like this:

1. Reader sees `-1` from `readingChan`.
  2. Reader calls `cancel()`.
  3. The loop loops back around.
  4. The `select` statement sees `case <-ctx.Done()` is ready.
  5. The loop exits.

Here is the updated `go func` block incorporating the context:

```
for {
    select {
        case <-ctx.Done():
            // CLEANUP ON EXIT
            r.sendStageCompletion(-1)
            for _, c := range tupleClients {
                c.Conn.Close()
            }
            _ = inputFile.Close()
            return
    }

    case tuple := <-readingChan:
        if tuple.lineNum == -1 {
            // Signal that we are done.
            // The loop will catch <-ctx.Done() on the next iteration (or immediately).
            cancel()
            continue
        }

        // ... (Your topology update logic) ...

        // ... (Your connection logic) ...

        // ... (Your send logic) ...

        // Rate Limiting Sleep (Context Aware)
        expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.S))
        targetTime := startTime.Add(expectedDuration)
        now := time.Now()

        if targetTime.After(now) {
            select {
                case <-time.After(targetTime.Sub(now)):
                    // sleep done
                case <-ctx.Done():
                    // cancelled while sleeping
                    cancel() // Ensure it's cancelled (idempotent)
                    continue // Loop around to hit the main ctx.Done case
            }
        }
        numProcessed++
    }
}

}()
```

## Next Step

Would you like me to update the full `main` function with this `context` pattern and the `RateLog` implementation we discussed, so you have a complete, clean version?

**User**

how does this look so far, I have added a logger goroutine to removing locking everytime to write to the log file:

```
package main

import (
    "bufio"
    "context"
    "fmt"
    "g14-mp4/RainStorm/resources"
    "g14-mp4/mp3/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "sort"
    "strconv"
    "strings"
    "sync"
    "time"
)
type WorkerIps struct {
    ips []net.IP
    sync.RWMutex
}

type RainStorm struct {
    NumStages int
    NumTasksPerStage int
    HydfsSrcDirectory string
    HydfsDestinationFileName string
    ExactlyOnce bool
    AutoScale bool
    InputRate float64
    LowestRate float64
    HighestRate float64
    Ops []Operation
    TaskInformation []map[int]*TaskInfo
    map[int]map[int]*TaskInfo
    stage int
    task int
    IP net.IP
    NextTaskNum int
    int
    NextAvailableVM int
    Stage1UpdatesChan chan map[int]net.IP
    Lock *sync.RWMutex
    DoneReading bool
    StartTime time.Time
   LogFile *os.File
    LogFileChan chan string
}
const clientTimeout = time.Second * 3

var workers WorkerIps
var numWorkers int
var numSuccessfulDials int
var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex
var appCompletedChan chan bool
var dataDir string

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        err := rpc.Register(&workers)
        if err != nil {
            fmt.Println(err)
            return
        }
        listener, err := net.Listen("tcp", "IntroducePort")
        if err != nil {
            fmt.Println(err)
            return
        }
        rpc.Accept(listener)()
        input := make(chan RainStorm)
        go processStdin(input)
        hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
        if err != nil {
            fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error())
        }
        defer hydfsClient.Close()
        for {
            r := <-input
            if numWorkers == 0 {
                panic("No workers")
            }
            r.LogFileChan = make(chan string, 100)
            r.StartTime = time.Now()
            //logger
            go func() {
                path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405"))
                r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
                r.LogFile.WriteString("Started RainStorm Application\n")
                select {
                    case <-ctx.Done():
                        r.LogFile.Close()
                        return
                    case s := <-r.LogFileChan:
                        fmt.Fprintf(r.LogFile, "RainStorm Application Completed")
                }
            }()
            // INITIATE NEW RAINSTORM APPLICATION
            // Global RM
            // 1. open listener for current task input rates from workers
            // 2. check if autoscale is on, if it is ->
            // 3. compare rates to see if changes are needed
            // 4. complete changes
            appServer := rpc.NewServer()
            err = appServer.Register(&r)
            if err != nil {
                fmt.Println(err)
                continue
            }
            globalRmListener, err := net.Listen("tcp", "GlobalRMPort")
            if err != nil {
                fmt.Println("GlobalRM unable to connect to worker: " + err.Error())
                continue
            }
            go func() {
                for {
                    conn, err := globalRmListener.Accept()
                    if err != nil {
                        if strings.Contains(err.Error(), "use of closed network connection") {
                            return
                        }
                        fmt.Println("GlobalRM Accept error: " + err.Error())
                    }
                    r.Lock.Lock()
                    rpcWorkers = make(map[string]*rpc.Client)
                    numSuccessfulDials = 0
                    rpcWorkersLock.Lock()
                    for _, workerIp := range workers.ips {
                        // collect list of tasks for this worker
                        err := rpc.Dial("tcp", workerIp.String() + AssignmentPort)
                        if err != nil {
                            fmt.Println("Unable to connect to worker: " + err.Error())
                            continue
                        }
                        rpcWorkers[workerIp.String()] = worker
                        numSuccessfulDials++
                    }
                    workers.Lock.Unlock()
                    rpcWorkersLock.Unlock()
                    r.Lock.Unlock()
                    r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
                    r.NextTaskNum = make([]int, r.NumStages)
                    r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
                    r.DoneReading = false
                    r.TaskCompletion = make([]CompletionTuple, r.NumStages)
                    r.initWorker()
                    r.NextAvailableVM = 0
                    for i := range r.NumStages {
                        r.TaskInformation[i] = make(map[int]*TaskInfo)
                    }
                    r.TaskCompletion[i] = CompletionTuple{
                        Counter: 0,
                        StateTracker: make(map[int]bool),
                    }
                    for j := range r.NumTasksPerStage {
                        r.addTask(i, j)
                        r.NextTaskNum[i]++
                    }
                    r.sendIps()
                    r.Lock.Unlock()
                }
            }()
            // TODO: read srcFile from HyDFS and send into system at Input Rate for this application
            // send stage -1 is done once done reading from the file
            // read in from local; output on HyDFS
            var createReply []resources.AddFileReply
            err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content: make([]byte, 0),
            }, &createReply)
            if err != nil {
                fmt.Println(err.Error())
            }
        }
    }()
}
```

```

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " + err.Error()) }

// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0] {
    firstStageTasks[tid] = info.Ip
    firstTaskList = append(firstTaskList, tid)
} sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct {
    line string
    lineNum int
}, 100) go func() {
    lineNum := 0
    for scanner.Scan() {
        readingChan <- struct {
            line string
            lineNum int
        }{
            line: scanner.Text(),
            lineNum: lineNum
        }
    }
    lineNum++
}
readingChan <- struct {
    line string
    lineNum int
}{"line": "", "lineNum": -1}
r.Lock.Lock()
r.DoneReading = true
r.Lock.Unlock()
}()
eofReceived := false
done := false
for {
    select {
        case tuple := <-readingChan:
            if tuple.lineNum == -1 {
                //done reading
                eofReceived = true
                continue
            }
            select {
                case updatedMap := <-r.Stage1UpdatesChan:
                    firstStageTasks = updatedMap
                    firstTaskList = make([]int, 0)
                    for k := range firstStageTasks {
                        firstTaskList = append(firstTaskList, k)
                    }
                    sort.Ints(firstTaskList)
                    default:
                }
            }
            nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
            nextTaskIp := firstStageTasks[nextTask]
            client, ok := tupleClients[nextTaskIp.String()]
            if !ok {
                conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
                if err != nil {
                    fmt.Println("Unable to connect to worker: " + err.Error())
                    delete(tupleClients, nextTaskIp.String())
                    continue
                }
                client = &WorkerClient{
                    Conn: conn,
                    Buf: bufio.NewReader(conn),
                }
                tupleClients[nextTaskIp.String()] = client
            }
            // Send the tuple
            _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
            // Id-Id, stage, task, data
            _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
            // Wait for the ack
            _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
            ack, err := client.Buf.ReadString('\n')
            expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
            if err != nil || strings.TrimSpace(ack) != expectedAck {
                client.Conn.Close()
                delete(tupleClients, nextTaskIp.String())
                readingChan <- tuple
                continue
            }
            expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second))
            targetTime := startTime.Add(expectedDuration)
            now := time.Now()
            if targetTime.After(now) {
                // ahead of schedule, sleep to sync with desired rate
                time.Sleep(targetTime.Sub(now))
            }
            default:
                // reached if channel is empty if eofReceived { done = true } } if done { break } }
            r.sendStageCompletion(-1)
            for _, c := range tupleClients {
                c.Conn.Close()
            }
            println("done closing conns")
            _ = inputFile.Close()
        readingChan := make(chan string, 200)
        go func() {
            buffer := make([]byte, 4096)
            for {
                line := <-readingChan
                buffer = append(buffer, []byte(line)...)
            }
            var reply []resources.AppendReply
            hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content: buffer,
            }, &reply)
        }()
        //listen for tuples to print to console and buffered
        appendToHydfs := func() {
            tupleListener, err := net.Listen("tcp", TuplePort)
            if err != nil {
                return
            }
            defer func() {
                tupleListener.Close()
            }()
            for {
                conn, err := tupleListener.Accept()
                if err != nil {
                    continue
                }
                go func(conn net.Conn) {
                    defer conn.Close()
                    reader := bufio.NewReader(conn)
                    for {
                        line, err := reader.ReadString('\n')
                        if err != nil {
                            return
                        }
                        fmt.Println(line)
                    }
                }(conn)
            }
        }()
        // needs to wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan //blocking
        println("RainStorm Application completed!")
        // CLEANUP: do once the current RainStorm application is done
        close(r.LogFileChan)
        cancel()
        rpcWorkersLock.Lock()
        for _, worker := range rpcWorkers {
            worker.Close()
        }
        rpcWorkersLock.Unlock()
        err = globalRmListener.Close()
        if err != nil {
            fmt.Println(err)
        }
    }
}

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error {
    workers.I.Lock()
    defer workers.I.Unlock()
    workers.ips = append(workers.ips, args)
    numWorkers++
    return nil
}

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    // restart the task on the next worker in the cycle
    app.Lock.Lock()
    defer app.Lock.Unlock()
    if _, exists := app.TaskInformation[task.Stage][task.TaskNumber];
        !exists {
            fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
        } else {
            workers.I.RLock()
            app.TaskInformation[task.Stage][task.TaskNumber].Ip =

```

```

workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply, string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs when receiving a tuple rate //app.LogFile if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f", args.Stage, args.Rate)
app.addTask(args.Stage, taskNum) app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager --> manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[args.Stage][args.Task]; exists { delete(app.TaskInformation[args.Stage], args.Task)
app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: %d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage][args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send completed stagID to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveIPs", app.TaskInformation, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile: app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.Initialize", args, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%numWorkers]} workers.l.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()]
rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err

```

```

!= nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply // @TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)))
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": // @TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

This looks much more robust! By moving the logging to a dedicated goroutine and using a channel, you've removed disk I/O from the RPC critical path.

However, I've identified **two specific bugs** in your logger and your input loop that will cause your application to hang or panic.

### 1. The "Single-Shot" Logger Bug

**The Problem:** In your logger goroutine, you used a `select` statement that handles **one** event and then exits.

Go



```

select {
case <-ctx.Done():
    // ... close ...
    return
case s := <-r.LogFileChan:
    _, _ = fmt.Fprintf(r.LogFile, s)
}
// This goroutine ends here!

```

**Result:** After the very first log message arrives, the goroutine finishes. All subsequent sends to `LogFileChan` will eventually fill the buffer (100) and then **deadlock your entire Leader**.

**The Fix:** Wrap the `select` in a `for` loop so it continues processing logs until finished.

Go



```

// logger logic inside main()
go func() {
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("200601
    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
    _, _ = r.LogFile.WriteString("Started RainStorm Application\n")

    // Use bufio for efficiency
    writer := bufio.NewWriter(r.LogFile)

    for {
        select {
        case <-ctx.Done():
            writer.WriteString("RainStorm Application Completed")
            writer.Flush()
            _ = r.LogFile.Close()
            return
        case s, ok := <-r.LogFileChan:
            if !ok { return } // Channel closed
            writer.WriteString(s)
            writer.Flush() // Ensure realtime logging
        }
    }
}()

```

## 2. The Divide-by-Zero Startup Panic

**The Problem:** As discussed, when your leader starts, `firstTaskList` is briefly empty before tasks initialize or if a scale-down happens.

Go



```
nextTask := firstTaskList[tuple.lineNum % len(firstTaskList)]
```

**Result:** If `len` is 0, this causes a **runtime panic**.

**The Fix:** Add the guard check inside your input loop:

```
Go
```

```
// Inside input loop selective logic:
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}
```



### 3. Structural Cleanup

The way you call `cancel()` and `close(r.LogFileChan)` needs to be slightly safer.

- You should `close(r.LogFileChan)` **first**, wait for the logger to finish draining, and then let the goroutine exit.
- Right now, if you `cancel()` and return immediately, the log file might be closed before the last few buffered log entries are written.

### Final Verification of the "Done Reading" Logic

You are using `r.DoneReading` to stop sending stage updates. This is safe, but make sure that in `ReceiveFailure`, you are using a **non-blocking send** to the update channel. You updated your code to send updates manually, but strictly speaking, if `Stage1UpdatesChan` fills up (buffer is 20), `ReceiveFailure` will block.

**Recommended change for `ReceiveFailure`:**

```
Go
```

```
if task.Stage == 0 && !app.DoneReading {
    temp := make(map[int]net.IP)
    for t, info := range app.TaskInformation[0] {
        temp[t] = info.Ip
    }
    // Use non-blocking select to ensure ReceiveFailure never hangs leader
    select {
        case app.Stage1UpdatesChan <- temp:
        default:
    }
}
```



**Would you like me to provide a clean refactor of the `main()` loop incorporating these stability fixes for the logger and the divide-by-zero guard?**

## User

for this when the listener is closed, do the connections also close or do I have to manually close them: for {  
select { case <-ctx.Done(): \_ = tupleListener.Close() return default: conn, err := tupleListener.Accept() if err != nil { continue } go func(conn net.Conn) { defer conn.Close() reader := bufio.NewReader(conn) for { line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed } fmt.Println(line) readingChan <- line } } (conn) } }

## Gemini

Closing the `net.Listener` does **not** close the active connections that have already been established.

- `tupleListener.Close()` only stops the server from accepting *new* connections.
- Existing connections (handled by your `go func(conn net.Conn)`) will remain open and active until you explicitly close them or the remote side disconnects.

To cleanly shut everything down, you need to track the active connections and close them manually when `ctx.Done()` triggers.

## The Solution: Connection Tracking

You need a simple way to register each connection when it opens and deregister it when it closes.

Go



```
go func() {
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        return
    }

    // 1. Create a container to track active connections
    var conns sync.Map // Thread-safe map

    // Ensure listener closes on exit
    defer func() {
        tupleListener.Close()

        // 2. FORCE CLOSE all active connections
        conns.Range(func(key, value interface{}) bool {
            if c, ok := key.(net.Conn); ok {
                c.Close()
            }
            return true
        })
    }()
}

// Run the Accept loop in a separate goroutine so we can use select on ctx.Done()
// OR just rely on Close() causing Accept() to fail.

// Simplest pattern:
```

```
go func() {
    <-ctx.Done()
    tupleListener.Close() // This breaks the Accept loop below
}()

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        // If listener was closed by ctx.Done(), we land here.
        return
    }

    // 3. Register the connection
    conns.Store(conn, true)

    go func(c net.Conn) {
        // 4. De-register when this specific handler finishes
        defer func() {
            conns.Delete(c)
            c.Close()
        }()

        reader := bufio.NewReader(c)
        for {
            line, err := reader.ReadString('\n')
            if err != nil {
                return // connection closed/failed
            }
            fmt.Println(line)

            // Check context before sending to avoid blocking on exit
            select {
            case readingChan <- line:
            case <-ctx.Done():
                return
            }
        }
    }(conn)
}

}()


```

## Why this is necessary

If you don't close the active connections manually:

1. **Resource Leak:** The file descriptors for those sockets remain open.

2. **Hanging Clients:** The workers connected to this leader will think the connection is still alive and might keep trying to send data into a "zombie" socket, delaying their own error handling/reconnection logic.
3. **Blocked Goroutines:** Your `reader.ReadString('\n')` will block forever if the client is idle, meaning those goroutines never exit, causing a memory leak. Calling `c.Close()` forces `ReadString` to return an error immediately, cleaning up the goroutine.

**User**

here is the entire file, check if this is necessary, also cant i just close the connections when I receive a ctx.done:

```

package main

import ( "bufio" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net" "net/rpc"
"os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.File LogFileChan chan string }
const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1) ctx, cancel := context.WithCancel(context.Background())
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HyDFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") }
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now()
//logger go func() { path := filepath.Join(homeDir, "RainStormLogs",
"RainStorm_" + r.StartTime.Format("20060102150405")) r.LogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString("Started RainStorm
Application\n") writer := bufio.NewWriter(r.LogFile) for { select { case <-ctx.Done():
writer.WriteString("RainStorm Application Completed") writer.Flush() _ = r.LogFile.Close() return case s, ok := <-r.LogFileChan: if !ok { //channel closed continue } writer.WriteString(s) writer.Flush() } } }() // INITIATE NEW
RAINSTORM APPLICATION //Global RM /* 1. open listener for current task input rates from workers 2. check if
autoscale is on, if it is -> 3. compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer()
err = appServer.Register(&r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go func() { for { conn, err := globalRmListener.Accept() if err != nil { if strings.Contains(err.Error(),
"use of closed network connection") { return // Exit quietly } fmt.Println("GlobalRM Accept error: " + err.Error())
return } //give connection to RPC server go appServer.ServeConn(conn) } }()
workers.l.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.l.RUnlock()
rpcWorkersLock.Unlock()

r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] = make(map[int]*TaskInfo)
//r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for j := range
r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps() r.Lock.Unlock()

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {

```

```

fmt.Println(err.Error()) }

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " + err.Error()) } //buffered write to HyDFS output file
readingChan := make(chan string, 200) go
func() { buffer := make([]byte, 4096) for { line := <-readingChan buffer = append(buffer, []byte(line...)) var
reply []resources.AppendReply _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: buffer, }, &reply) } }()
//listen for tuples to print to console and buffered append to hydfs go func() { tupleListener, err :=
net.Listen("tcp", TuplePort) if err != nil { return } for { select { case <-ctx.Done(): _ = tupleListener.Close()
return default: conn, err := tupleListener.Accept() if err != nil { continue } go func(conn net.Conn) { defer
conn.Close() reader := bufio.NewReader(conn) for { line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed } fmt.Println(line) readingChan <- line } }(conn) } } }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }() eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } println("done closing conns") _ =
inputFile.Close() }() // needs to wait for the application to complete before cleaning up --> @TODO: come back
to this <-appCompletedChan //blocking println("RainStorm Application completed!") // CLEANUP: do once the
current RainStorm application is done cancel() close(r.LogFileChan) rpcWorkersLock.Lock() for _, worker :=
range rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }

}

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==

```

```

0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply, string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d", args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.addTask(args.Stage, taskNum) app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager --> manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[args.Stage][args.Task]; exists { delete(app.TaskInformation[args.Stage], args.Task) app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: %d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage][args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT should not have received this\n", args.Task) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send completed stagelD to workers: " + x.Error.Error()) } }
}

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs", app.TaskInformation, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile: app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }

func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.I.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%numWorkers]} workers.I.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err

```

```

!= nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply // @TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)))
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": // @TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

You are extremely close to a fully robust production-ready system.

To answer your specific question: **Yes, you absolutely need to close the connections manually.** If you just
close the `Listener`, the active clients stay connected forever (until they timeout or the OS kills them).

Here is the breakdown of the final critical fixes required in your provided code, specifically addressing your
listener logic and the lingering fatal bugs.

### 1. Fix the Tuple Listener (Connection Leak)

Your current code waits for `ctx.Done()`, closes the listener, and returns. **This leaks every single active
connection**, leaving the goroutines running forever.

**Correct Implementation:** You need to track connections and close them.

Go



```
// Inside your main loop...
// Listen for tuples to print to console
go func() {
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        return
    }

    // MAP TO TRACK ACTIVE CONNECTIONS
    var activeConns sync.Map

    // CLEANUP ROUTINE
    go func() {
        <-ctx.Done()
        tupleListener.Close() // Stop accepting new ones

        // FORCE CLOSE ALL EXISTING ONES
        activeConns.Range(func(key, value interface{}) bool {
            if conn, ok := key.(net.Conn); ok {
                conn.Close()
            }
            return true
        })
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        // Listener closed or error
        return
    }

    // REGISTER CONNECTION
    activeConns.Store(conn, true)

    go func(c net.Conn) {
        defer func() {
            c.Close()
            activeConns.Delete(c) // Deregister
        }()
    }

    reader := bufio.NewReader(c)
```

```

        for {
            line, err := reader.ReadString('\n')
            if err != nil {
                return
            }

            // Non-blocking send to avoid hanging on exit
            select {
                case readingChan <- line:
                case <-ctx.Done():
                    return
                }
            }
        }(conn)
    }
}()
```

## 2. Fix the "Divide By Zero" Panic (Critical)

You still have not added the check for empty task lists. **Your code will crash 100% of the time on startup.**

**In your input loop:**

Go



```

// ... inside select case updatedMap ...
sort.Ints(firstTaskList)
// ...

// ADD THIS CHECK NOW:
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}

// Then do the math:
nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
```

## 3. Fix the Deadlock (Leader Freeze)

You are still using blocking sends in `addTask`, `removeTask`, and `ReceiveFailure`. If the input loop is busy (e.g. handling a timeout), your Leader will hang and stop processing RPCs.

**Change this block in all 3 functions:**

Go



```
// REPLACE THIS:  
// app.Stage1UpdatesChan <- temp  
  
// WITH THIS:  
select {  
case app.Stage1UpdatesChan <- temp:  
default:  
    // Drop update to prevent deadlock.  
    // Safe because the next update will have the correct state anyway.  
}
```

## 4. Fix the Logger Goroutine (Exit Bug)

Your logger currently exits after writing **one** line because you missed the loop inside the select.

**Current (Buggy):**

```
Go  
  
// Your code:  
for {  
    select {  
        case <-ctx.Done():  
            // ... close and return ...  
        case s, ok := <-r.LogFileChan:  
            if !ok { continue } // BUG: This spins forever consuming CPU if channel closed  
            writer.WriteString(s)  
            writer.Flush()  
    }  
    // Loop continues  
}
```

**Corrected:**

```
Go  
  
// Logger  
go func() {  
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("200601  
    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)  
    _, _ = r.LogFile.WriteString("Started RainStorm Application\n")  
    writer := bufio.NewWriter(r.LogFile)  
  
    for {  
        select {  
            case <-ctx.Done():
```

```
writer.WriteString("RainStorm Application Completed")
writer.Flush()
_ = r.LogFile.Close()
return

case s, ok := <-r.LogFileChan:
    if !ok {
        return // Exit if channel is closed
    }
    writer.WriteString(s)
    writer.Flush()
}

}
}

}()
```

## Summary

1. **Listener:** Use `sync.Map` to track and force-close connections on `ctx.Done`.
2. **Panic:** Add `if len(firstTaskList) == 0` check.
3. **Deadlock:** Use `select { case ch<-x: default: }` for updates.
4. **Logger:** Ensure `case s, ok := ...` returns if `!ok`.

Once these 4 changes are made, your Leader code is robust.

**User**

to close the connections, cant I just add wait in the goroutine for a ctx.done and close it then

**Gemini**

The short answer is: **Yes, but you cannot do it inside the reading goroutine.**

You cannot write `select { case <-ctx.Done(): ... }` around `reader.ReadString('\n')` because `ReadString` is a **blocking call**. If the worker is idle (not sending data), your code is stuck frozen on that line. It will not "wake up" to check `ctx.Done()` until *after* data arrives.

However, you **can** achieve your goal without a global map by launching a tiny "**Sidecar" Goroutine** for each connection. This separate goroutine does exactly what you asked: it waits for `ctx.Done()` and immediately kills the connection.

## The "Sidecar" Pattern

This is the cleanest way to handle this without global maps or complex loops.

1. **Reader Goroutine:** blocked on `ReadString`.
2. **Closer Goroutine:** waits on `ctx.Done`. When it triggers, it calls `conn.Close()`.
3. **Result:** The `Close()` call forces the `ReadString` to fail immediately with an error, unblocking the Reader and allowing it to exit.

Here is the implementation for your `TupleListener`:

Go



```
// listen for tuples to print to console
go func() {
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        return
    }

    // 1. Routine to close Listener on exit
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        return // Listener closed
    }

    // 2. The "Sidecar" Closer Routine
    // This waits for context cancellation and KILLS this specific connection
    go func(c net.Conn) {
        <-ctx.Done()
        c.Close()
    }(conn)
}
```

```

        c.Close() // This forces the Read below to error out immediately
    }(conn)

    // 3. The Reader Routine
    go func(c net.Conn) {
        // Ensure we close if we exit normally (e.g. read error)
        defer c.Close()

        reader := bufio.NewReader(c)
        for {
            line, err := reader.ReadString('\n')
            if err != nil {
                return // Connection closed by Sidecar or remote end
            }

            fmt.Println(line) // Print to console

            // Non-blocking send to file writer
            select {
            case readingChan <- line:
            case <-ctx.Done():
                return
            }
        }
    }(conn)
}
}()
```

## Why this is better

- **No Global Locks/Maps:** You don't need `sync.Map` to track connections.
- **Immediate Cleanup:** As soon as you call `cancel()` in main, the Sidecar fires, closes the socket, and the Reader wakes up and exits.

## ⚠ FINAL REMINDER: The Startup Crash

I noticed you still haven't added the fix for the **Divide By Zero** panic in your input loop. Your code **will crash** on startup without this check.

Please paste this into your `processStdin` loop immediately:

Go



```

// Inside input loop...
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}
```

```
}
```

```
// nextTask := ...
```

**User**

I have pasted my updated code below. Why am I not printing out the lines when listening for workers sending tuples:

```
package main

import ("bufio" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net" "net/rpc"
"os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.File LogFileChan chan string }
const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1) ctx, cancel := context.WithCancel(context.Background())
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }() input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HyDFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") }
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now()
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { return } //logger go func() { path :=
filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405")) r.LogFile, _ =
os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString("Started
RainStorm Application\n") writer := bufio.NewWriter(r.LogFile) for { select { case <-ctx.Done():
writer.WriteString("RainStorm Application Completed") writer.Flush() _ = r.LogFile.Close() return case s, ok :=
<-r.LogFileChan: if !ok { //channel closed continue } writer.WriteString(s) writer.Flush() } } }()
// INITIATE NEW
RAINSTORM APPLICATION //Global RM /* 1. open listener for current task input rates from workers 2. check if
autoscale is on, if it is -> 3. compare rates to see if changes are needed 4. complete changes */ appServer :=
rpc.NewServer() err = appServer.Register(&r) if err != nil { fmt.Println(err) continue } globalRmListener, err :=
net.Listen("tcp", GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error())
continue } go func() { for { conn, err := globalRmListener.Accept() if err != nil { if strings.Contains(err.Error(),
"use of closed network connection") { return // Exit quietly } fmt.Println("GlobalRM Accept error: " + err.Error())
return } //give connection to RPC server go appServer.ServeConn(conn) } }()
workers.l.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.l.RUnlock()
rpcWorkersLock.Unlock()

r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] = make(map[int]*TaskInfo)
//r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for j := range
r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps() r.Lock.Unlock()

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
```

```

fmt.Println(err.Error()) }

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open src directory: " + err.Error()) } //buffered write to HyDFS output file
readingChan := make(chan string, 200) go
func() { buffer := make([]byte, 4096) for { line := <-readingChan buffer = append(buffer, []byte(line...)) var
reply []resources.AppendReply _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: buffer, }, &reply) } }()
//listen for tuples to print to console and buffered append to hydfs go func() { for { select { case <-ctx.Done():
_ = tupleListener.Close() return default: conn, err := tupleListener.Accept() if err != nil { continue } go
func(conn net.Conn) { defer conn.Close() reader := bufio.NewReader(conn) for { select { case <-ctx.Done():
return default: line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed }
fmt.Println(line) readingChan <- line } } }(conn) } }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }() eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } println("done closing conns") _ =
inputFile.Close() }() // needs to wait for the application to complete before cleaning up --> @TODO: come back
to this <-appCompletedChan //blocking println("RainStorm Application completed!") // CLEANUP: do once the
current RainStorm application is done cancel() close(r.LogFileChan) rpcWorkersLock.Lock() for _, worker :=
range rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }

}

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==

```

```

0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply, string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d", args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.addTask(args.Stage, taskNum) app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager --> manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: %d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage][args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage], args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send completed stagelD to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs", app.TaskInformation, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile: app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.I.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%numWorkers]} workers.I.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err

```

```

!= nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply // @TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)))
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": // @TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

The reason you are not seeing any output is most likely due to **Silent Failure** in your error handling.

In your current code, if `net.Listen`, `Accept`, or `ReadString` fails, you simply `return` or `continue` without printing the error. If the port is busy (common on restarts) or the worker disconnects immediately, the code exits silently, and you see nothing.

Additionally, regarding your question about closing connections: **No, you cannot simply wait for `ctx.Done` in the same goroutine.** `reader.ReadString('\n')` is a **blocking call**. If the worker stops sending data, your code freezes on that line. It will never reach the `case <-ctx.Done()` check. You need a **Sidecar Goroutine** to kill the connection from the outside.

Here is the fixed `TupleListener` block. Replace your current `go func()` (lines 262-290) with this robust version:

## The Fix

Go



```
// Listen for tuples to print to console and buffered append to hydfs
go func() {
    // 1. Log errors if Listen fails (e.g., Address already in use)
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        fmt.Println("Error listening on TuplePort:", err)
        return
    }

    // Ensure listener closes when context cancels
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()

    for {
        // 2. Remove the 'select' here. Accept() is blocking, which is fine.
        // If ctx.Done() closes the listener, Accept() returns an error, breaking the
        conn, err := tupleListener.Accept()
        if err != nil {
            // Determine if this is a normal shutdown or a real error
            select {
                case <-ctx.Done():
                    return // Normal shutdown
                default:
                    fmt.Println("Error accepting tuple connection:", err)
                    continue
            }
        }

        // Handle the connection
        go func(c net.Conn) {
            // 3. THE SIDECAR: This waits for context and kills the connection
            // This forces ReadString to unblock immediately with an error if we shut down
            go func() {
                <-ctx.Done()
                c.Close()
            }()
            defer c.Close()
            reader := bufio.NewReader(c)
        }
    }
}
```

```

        for {
            // 4. Just read. If Sidecar kills connection, this returns error.
            line, err := reader.ReadString('\n')
            if err != nil {
                // Log the error so you know WHY it stopped (EOF, Reset, etc)
                if err.Error() != "EOF" && !strings.Contains(err.Error(), "use of clo
                    fmt.Println("Tuple read error:", err)
                }
                return
            }

            fmt.Println(line)

            // Non-blocking send to prevent hanging if HyDFS is slow
            select {
                case readingChan <- line:
                case <-ctx.Done():
                    return
                }
            }
        }(conn)
    }()
}

```

## Key Changes Explained

### 1. The Sidecar (`go func() { <-ctx.Done(); c.Close() }`):

- This runs in parallel with your reader.
- If `ReadString` is stuck waiting for a worker to send data, and you hit `Ctrl+C` (or finish the app), this sidecar closes the socket.
- Closing the socket forces `ReadString` to wake up with an error, allowing the goroutine to exit cleanly.

### 2. Verbose Error Logging:

- I added `fmt.Println` to the `err != nil` blocks.
- **Scenario:** If you restart your app quickly, the OS might hold onto the `TuplePort` for a few seconds (`TIME_WAIT`). Your original code would fail `net.Listen` and exit silently. This new code will tell you: “*address already in use*”.

### 3. Removed the `select { default: ReadString }` Anti-pattern:

- Checking `ctx.Done` before reading creates a race condition where you might block on read right after the check. The Sidecar pattern is the only 100% reliable way to cancel a blocking network read in Go.

## User

I modified this and saw that the error that it is receiving an eof:

```
default: line, err := reader.ReadString('\n') if err != nil { println("Error when reading from tuple port:" + err.Error()) return // connection closed/failed } fmt.Println(line) readingChan <- line }
```

Gemini

The `EOF` error confirms that the connection is successfully established, but the **Worker closes it immediately** without sending any data.

This typically happens for one of two reasons in your specific architecture:

1. **The Worker Crashed/Exited:** The worker process dialled the Leader, but then crashed or finished its task immediately and closed the socket.
  2. **Protocol Mismatch (The Likely Cause):** The Worker is expecting the Leader to say something *first* (handshake), or the Worker is sending data that doesn't end in `\n`, causing `ReadString` to hang until the socket closes.

## Diagnosis & Fix

Since you are implementing a Stream Processing system, the connection flow usually works like this:

- **Leader:** Listens on `TuplePort`.
  - **Worker:** Dials `TuplePort`.
  - **Worker:** Sends Tuples (`key,value\n`).

If you get an EOF immediately, check your **Worker Code**.

## Scenario A: The Worker is "One-Shot"

If your worker code looks like this:

Go

1

```
// Worker Code  
conn, _ := net.Dial("tcp", leaderIP + TuplePort)  
fmt.Fprintf(conn, "some data")  
conn.Close() // <--- Closes immediately!
```

If "some data" does **not** end with `\n`, the `Leader's ReadString('\'\n')` will wait for a newline that never comes. Eventually, it hits the `Close()`, which triggers the `EOF` error.

**Fix:** Ensure the worker sends a newline:

Go

1

```
fmt.Fprintf(conn, "some data\n") // Add \n
```

## **Scenario B: The Worker Logic Logic**

If the Worker connects but has nothing to send yet (e.g., waiting for its own input), and then shuts down, you will see an EOF.

**Fix:** You should treat `EOF` as a normal "Worker Disconnected" event, not a fatal error. Just return from the handler loop.

## Updated Code to Handle EOF Gracefully

You should modify your listener loop to ignore `EOF` (it's normal when a client disconnects) and print other errors.

Go



```
line, err := reader.ReadString('\n')
if err != nil {
    if err.Error() == "EOF" {
        // Normal behavior: Worker finished and closed connection
        return
    }
    // Abnormal behavior: Network error
    fmt.Println("Error reading tuple:", err)
    return
}
fmt.Println(line)
readingChan <- line
```

## Debugging Step

To know *exactly* what is happening, check your **Worker logs**.

1. Does the Worker successfully dial?
2. Does the Worker print "Sending tuple..."?
3. Does the Worker panic shortly after dialing?

If the Worker is crashing, the OS closes the socket, resulting in the `EOF` you are seeing on the Leader.

**User**

the line is actually printing though, so how can I handle the eof when it comes

**Gemini**

If the line **is** printing before the error appears, then your system is actually working correctly!

The `EOF` (End of File) error simply means the Worker finished sending its data and closed the connection.

**What is happening:**

1. **Read 1:** Leader reads "Hello World\n". Success. Prints it.
2. **Loop:** The code loops back to read again.
3. **Worker:** Has no more data, calls `conn.Close()`.
4. **Read 2:** Leader tries to read, sees the socket is closed, returns `EOF`.

**The Fix**

You should simply **catch and ignore** the EOF error, treating it as a signal to stop reading from that specific connection.

Replace your reading block with this:

Go



```
line, err := reader.ReadString('\n')
if err != nil {
    // EOF is normal. It just means the worker is done talking.
    if err.Error() == "EOF" {
        return
    }
    // Any other error is a real problem.
    fmt.Println("Error when reading from tuple port: " + err.Error())
    return
}
fmt.Println(line)
readingChan <- line
```

**User**

here is the updated file, why is it not writing to the log file when the application is completed, but the `println` does happen

```

package main

import ("bufio" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net" "net/rpc"
"os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.File LogFileChan chan string }
const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1) ctx, cancel := context.WithCancel(context.Background())
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HyDFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") }
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now()
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { return } //logger go func() { path :=
filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755) r.LogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString("Started RainStorm
Application\n") writer := bufio.NewWriter(r.LogFile) for { select { case <-ctx.Done():
writer.WriteString("RainStorm Application Completed") writer.Flush() _ = r.LogFile.Close() return case s, ok :=
<-r.LogFileChan: if !ok { //channel closed continue } writer.WriteString(s) writer.Flush() } } }()
// INITIATE NEW
RAINSTORM APPLICATION //Global RM /* 1. open listener for current task input rates from workers 2. check if
autoscale is on, if it is -> 3. compare rates to see if changes are needed 4. complete changes */ appServer := rpc.NewServer()
err = appServer.Register(&r) if err != nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp",
GlobalRMPort) if err != nil { fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go func() { for { conn, err := globalRmListener.Accept() if err != nil { if strings.Contains(err.Error(),
"use of closed network connection") { return // Exit quietly } fmt.Println("GlobalRM Accept error: " + err.Error())
return } //give connection to RPC server go appServer.ServeConn(conn) } }()
workers.l.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.l.RUnlock()
rpcWorkersLock.Unlock()

r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] = make(map[int]*TaskInfo)
//r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for j := range
r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps() r.Lock.Unlock()

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
```

```

RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) } //buffered write to HyDFS output file readingChan := make(chan string, 200) go
func() { buffer := make([]byte, 4096) for { line := <-readingChan buffer = append(buffer, []byte(line)...)} var
reply []resources.AppendReply _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: buffer, }, &reply) } }()
//listen for tuples to print to console and buffered append to hydfs go func() { for { select { case <-ctx.Done():
_ = tupleListener.Close() return default: conn, err := tupleListener.Accept() if err != nil { continue } go
func(conn net.Conn) { defer conn.Close() reader := bufio.NewReader(conn) for { select { case <-ctx.Done():
return default: line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed }
fmt.Println(line) readingChan <- line } } }(conn) } }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-r.Stage1UpdatesChan:
firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } _ = inputFile.Close() }()
// needs to
// wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan
//blocking println("RainStorm Application completed!") // CLEANUP: do once the current RainStorm application
is done cancel() close(r.LogFileChan) rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ =
worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }
}
func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =

```

```

workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage == 0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply, string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d", args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.addTask(args.Stage, taskNum) app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f", args.Stage, args.Rate) app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager --> manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: %d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage][args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage], args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { // stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs", app.TaskInformation, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile: app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED //if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%numWorkers]} workers.l.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]]

```

```
[taskNum].Ip.String()]) rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum][taskNum].Pid = reply // @TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply, string(app.Ops[stageNum].Name)) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoScaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case "RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch { case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args: splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i] break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i] break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: // AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true } break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": // @TODO: add implementation for this break
case "list_tasks": break
} } }
```

## Gemini

You are not seeing lines printed because you have **silenced all errors**.

In your current code:

```
Go
```

```
line, err := reader.ReadString('\n')
if err != nil {
    return // <--- YOU RETURN SILENTLY HERE
```

```
}
```

```
fmt.Println(line)
```

If the worker sends data that doesn't end in a newline, or if it disconnects ( EOF ), or if there is a network reset, your code returns immediately **without printing the error** and **without printing the line**.

Additionally, your implementation of `select { default: ReadString }` is **incorrect**. `ReadString` blocks the thread, so the `case <-ctx.Done():` check above it will **never run** until a message actually arrives.

Here is the corrected `TupleListener` block. I have added print statements so you can see *why* it is failing, and I restored the **Sidecar Pattern** because it is the only way to correctly cancel a blocking read.

## The Fix

Replace the entire `//listen for tuples... go func` (lines 230-258) with this:

```
Go
```

```
// Listen for tuples to print to console and buffered append to hydfs
go func() {
    // 1. Log if listening fails (e.g. port in use)
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        fmt.Println("Error listening on TuplePort:", err)
        return
    }

    // 2. Cleanup routine: Close listener when context is done
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()

    for {
        // 3. Accept is blocking. If listener closes, this returns error.
        conn, err := tupleListener.Accept()
        if err != nil {
            // Check if this is a normal shutdown
            select {
                case <-ctx.Done():
                    return
                default:
                    fmt.Println("Error accepting tuple connection:", err)
                    continue
            }
        }

        // Handle connection
        go func(c net.Conn) {

```

```

    // 4. Sidecar: Force close connection on context cancel
    // This unblocks ReadString immediately
    go func() {
        <-ctx.Done()
        c.Close()
    }()
    defer c.Close()

    reader := bufio.NewReader(c)
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            // 5. PRINT THE ERROR so you know what happened
            if err.Error() != "EOF" && !strings.Contains(err.Error(), "use of clo
                fmt.Println("Tuple read error:", err)
            }
            return
        }

        fmt.Println(line)

        // Non-blocking send
        select {
        case readingChan <- line:
        case <-ctx.Done():
            return
        }
    }
}(conn)
}
}()

```

## Checklist of other Fatal Errors still in your code

You still have not applied the critical fixes we discussed previously. Your code **will still crash** or **deadlock** unless you fix these two specific spots:

- 1. Startup Crash (Divide by Zero):** Inside your input loop (around line 186), you calculate `tuple.lineNum % len(firstTaskList)`. You must add this check before that line:

Go



```

if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
}

```

```
        continue  
    }
```

2. **Leader Freeze (Deadlock):** In `addTask`, `removeTask`, and `ReceiveFailure`, you are using blocking sends to `Stage1UpdatesChan`. **Change them to:**

Go



```
select {  
    case app.Stage1UpdatesChan <- temp:  
    default:  
}
```

**User**

no but it is printing, it is just when the application completes that it does not log it

**Gemini**

The issue is a **Race Condition** caused by how `select` works with a closed channel.

When you call `close(r.LogFileChan)` in your cleanup block, the `case s, ok := <-r.LogFileChan` becomes **permanently ready** (returning `ok=false`). Your code currently does this:

**Go**

```
if !ok {  
    // channel closed  
    continue // <--- THIS IS THE BUG  
}
```

This creates an **Infinite Loop**. The `select` keeps picking the closed channel case over and over again, spinning the CPU, and it potentially "starves" the `case <-ctx.Done()` branch so it never gets a chance to run.

**The Fix**

You should handle the shutdown in **both** cases. If the channel is closed (`!ok`), it means the application is done, so you should write the completion message and exit right there.

Replace your logger goroutine with this corrected version:

**Go**

```
// logger  
go func() {  
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("")  
    _ = os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)  
    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)  
    _, _ = r.LogFile.WriteString("Started RainStorm Application\n")  
  
    writer := bufio.NewWriter(r.LogFile)  
  
    // Ensure we close file on exit  
    defer func() {  
        writer.Flush()  
        r.LogFile.Close()  
    }()  
  
    for {  
        select {  
            case <-ctx.Done():  
                writer.WriteString("RainStorm Application Completed\n")  
        }  
    }  
}
```

```
        return // Defer triggers close

    case s, ok := <-r.LogFileChan:
        if !ok {
            // Channel closed. This is also a valid completion signal.
            writer.WriteString("RainStorm Application Completed\n")
            return // Defer triggers close
        }
        writer.WriteString(s)
        writer.Flush()
    }

}

}()
```

## ⚠ Final Warning: The Crash

You still have not fixed the **Divide By Zero** crash in your input loop. Your application **will panic** on startup.

Please add this check inside your input loop (`processStdin` -> `go func`):

Go



```
// Inside the input loop, BEFORE calculating nextTask
if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    readingChan <- tuple
    continue
}

// Then your existing line:
nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
```

**User**

does this look correct:

```
outputChan := make(chan string, 200)
go func() {
    buffer := bytes.Buffer{}
    for {
        select {
            case <-ctx.Done():
                if buffer.Len() > 0 {
                    var reply []resources.AppendReply
                    _ = hydfsClient.Call("Client.RemoteAppend",
                        &resources.RemoteFileArgs{
                            RemoteName: r.HydfsDestinationFileName,
                            Content:    buffer.Bytes(),
                        },
                        &reply)
                    buffer.Reset()
                }
            default:
                line := <-outputChan
                buffer.WriteString(line)
                if buffer.Len() > 4096 {
                    var reply []resources.AppendReply
                    _ = hydfsClient.Call("Client.RemoteAppend",
                        &resources.RemoteFileArgs{
                            RemoteName: r.HydfsDestinationFileName,
                            Content:    buffer.Bytes(),
                        },
                        &reply)
                    buffer.Reset()
                }
        }
    }
}
```

**Gemini**

This code has a **critical logic error** that will cause your application to hang during shutdown, and potentially another error regarding how `select` works.

**The Problem: Blocking inside `default`**

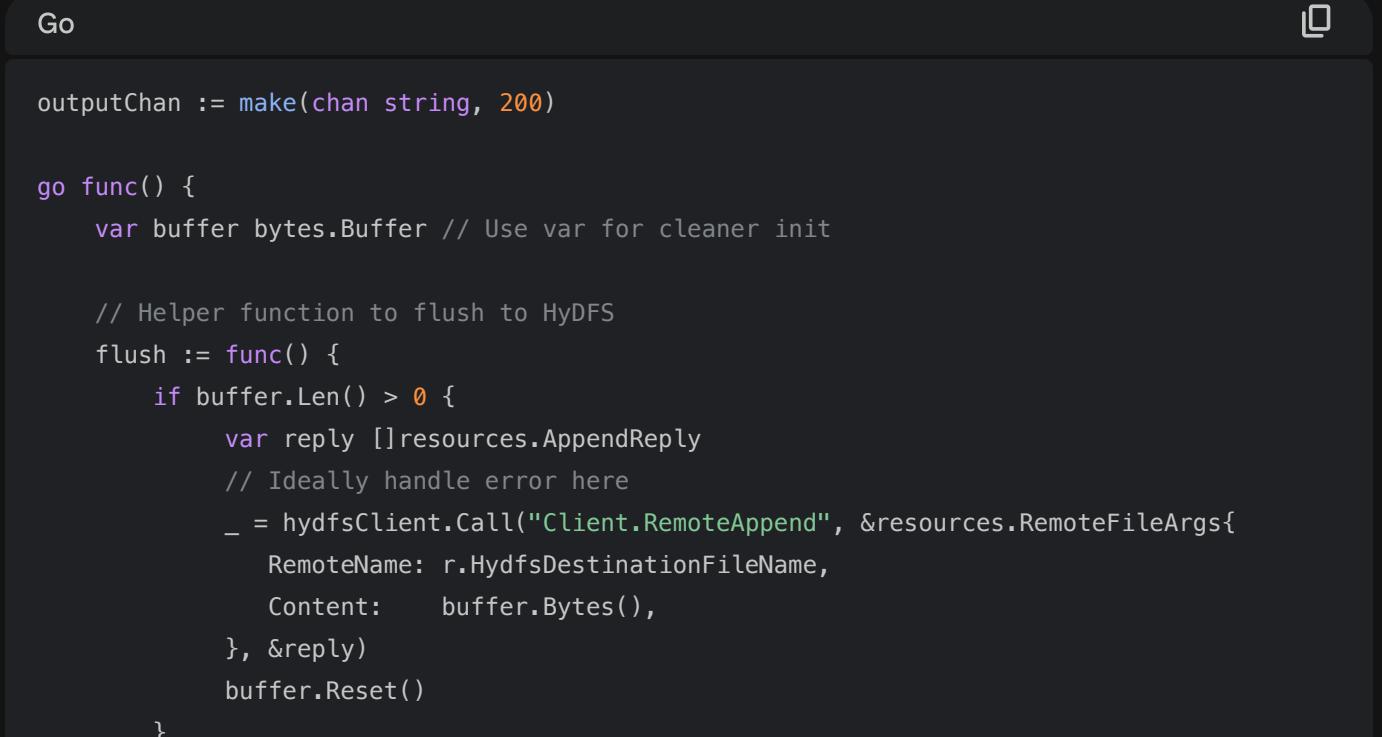
You placed the channel read `line := <-outputChan` inside the `default` case.

1. **It defeats the `select`:** The `select` statement checks if `ctx.Done()` is ready. If not, it **immediately** enters `default`.
2. **It hangs on Exit:** Once inside `default`, the code executes `line := <-outputChan`. This is a **blocking call**.
3. **The Deadlock:** If you want to shut down the app (cancel the context), but no new data is coming into `outputChan`, your goroutine is stuck frozen on that line inside `default`. It will **never** go back up to check `case <-ctx.Done()` until a new message arrives.

**The Fix**

You must make the channel read a **case** inside the `select`, not execute it inside `default`. This allows Go to wait on *either* a new line OR the shutdown signal simultaneously.

Here is the corrected version:



```
outputChan := make(chan string, 200)

go func() {
    var buffer bytes.Buffer // Use var for cleaner init

    // Helper function to flush to HyDFS
    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            // Ideally handle error here
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            },
            &reply)
            buffer.Reset()
        }
    }
}
```

```
}

for {
    select {
        case <-ctx.Done():
            // 1. Context cancelled? Flush remaining data and EXIT.
            flush()
            return

        case line := <-outputChan:
            // 2. Data received? Buffer it.
            buffer.WriteString(line)

            // 3. Buffer full? Flush.
            if buffer.Len() > 4096 {
                flush()
            }
        }
    }
}()
```

## Why this is correct

- 1. Responsive Shutdown:** If `ctx.Done()` triggers while you are waiting for data, the `select` will immediately pick that case, flush your data, and exit.
- 2. No Busy Waiting:** The `select` puts the thread to sleep until either data arrives or the context is cancelled, consuming 0% CPU while waiting.

**User**

```
what about now: outputChan := make(chan string, 200) go func() { buffer := bytes.Buffer{} for { select { case
<-ctx.Done(): if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } case line := <-outputChan:
buffer.WriteString(line) if buffer.Len() > 4096 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } } } }()
```

**Gemini**

This looks **almost perfect**. You correctly moved the channel read into the `select` case, which solves the blocking/hanging issue.

However, you have one **Fatal Bug** left: **You forgot to `return` inside the `ctx.Done()` case.**

**The Bug: Infinite Loop on Exit**

When `ctx.Done()` is closed, that case becomes "always ready".

1. The code enters `case <-ctx.Done():`.
2. It flushes the buffer.
- 3. It does NOT return.**
4. It loops back to the start.
5. `case <-ctx.Done()` is still ready, so it runs again immediately.
6. Your CPU hits 100% usage and the goroutine never actually exits.

**The Fix**

Add `return` after the flush logic.

Go



```
outputChan := make(chan string, 200)

go func() {
    buffer := bytes.Buffer{} // var buffer bytes.Buffer is slightly cleaner, but this works

    for {
        select {
        case <-ctx.Done():
            // 1. Flush remaining data
            if buffer.Len() > 0 {
                var reply []resources.AppendReply
                _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                    RemoteName: r.HydfsDestinationFileName,
                    Content:   buffer.Bytes(),
                }, &reply)
                buffer.Reset()
            }
        }
    }
}
```

```
    }

    // 2. CRITICAL: Exit the goroutine!
    return

    case line := <-outputChan:
        buffer.WriteString(line)
        if buffer.Len() > 4096 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
}

}()
```

**User**

```
for this go routine, why is it not sending at the desired input rate
go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock() firstStageTasks := make(map[int]net.IP)
firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0] { firstStageTasks[tid] = info.ip
firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock() tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed float64 = 0
readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for scanner.Scan() {
readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber} lineNumber++ }
readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true
r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime := startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } default: // reached if channel is empty if eofReceived { done = true } } if
done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } _ = inputFile.Close() }()
```

**Gemini**

There are **two distinct reasons** why your code is not respecting the `InputRate`. One is a logic bug that makes it go "too fast" (ignoring the limit), and the other is an architectural bottleneck that makes it go "too slow" (capping the max throughput).

**1. The Logic Bug: You never increment `numProcessed`**

In the code snippet provided, you initialized `var numProcessed float64 = 0`, but **you never increment it** inside the loop.

**The Consequence:**

- `expectedDuration` is calculated as `(0 / r.InputRate)`, which is always `0`.
- `targetTime` remains fixed at `startTime`.
- As soon as the program runs for a few milliseconds, `now` becomes greater than `startTime`.
- The check `if targetTime.After(now)` is always **false**.
- Result:** The code **never sleeps**. It ignores your rate limit entirely and sends tuples as fast as the network allows.

**The Fix:** Add `numProcessed++` at the end of your select block (after the successful Send/ACK).

Go



```
// ... inside the successful send block ...

if targetTime.After(now) {
    time.Sleep(targetTime.Sub(now))
}

numProcessed++ // <--- ADD THIS HERE
```

## 2. The Performance Bottleneck: Stop-and-Wait ACKs

Even if you fix the counter, your maximum throughput is severely limited by your **Synchronous ACK** logic.

**The Code causing the issue:**

Go



```
// 1. Send data
fmt.Fprintf(client.Conn, ...)

// 2. STOP and WAIT for round-trip confirmation
ack, err := client.Buf.ReadString('\n')
```

**The Math:**

- Distributed systems have latency (RTT). Even on a fast local network, RTT might be 0.5ms or 1ms.
- If RTT is 1ms, you can physically only perform **1,000 operations per second** per thread ( $1s/1ms = 1000$ ).
- If your `InputRate` is set to 5,000, you will never reach it. You will hit a hard ceiling at 1,000.

**The Fix:** If you need high throughput (e.g., >1,000 ops/sec), you must **pipeline** your requests.

1. **Split the Reader and Writer:** One goroutine sends tuples as fast as `InputRate` allows.
2. **Async ACKs:** A separate goroutine listens on the connection for ACKs.
3. **Windowing:** If `ExactlyOnce` is true, simply track "Pending ACKs" in a map. If the map gets too big, pause sending.

**Summary**

1. **If your observed rate is wildly high:** It is because you forgot `numProcessed++`.
2. **If your observed rate is stuck at a low ceiling:** It is because you are waiting for an ACK for every single message before sending the next one.

**User**

format time by month day hour minute second

**Gemini**

In Go, you format dates using a specific reference time: `Mon Jan 2 15:04:05 MST 2006`.

To format by **Month Day Hour Minute Second**, use this layout string:

Go



```
time.Now().Format("Jan 02 15:04:05")
```

If you prefer **numeric** month (e.g., `12-07 12:18:45`), use:

Go



```
time.Now().Format("01-02 15:04:05")
```

## Applied to your Logger

Update your logger line to include this timestamp:

Go



```
// ... inside the logger loop ...
case s, ok := <-r.LogFileChan:
    if !ok { return }

    // Format: Dec 07 12:18:45
    timestamp := time.Now().Format("Jan 02 15:04:05")

    // Result: [Dec 07 12:18:45] Rate: 500.00...
    line := fmt.Sprintf("[%s] %s\n", timestamp, s)

    writer.WriteString(line)
    writer.Flush()
```

**User**

also add the year

**Gemini**

To add the year, you simply include `2006` in your format string (since Go uses the specific date `2006-01-02` as the reference layout).

Here is the format string for **Month Day Year Hour:Minute:Second**:

**Go**

```
time.Now().Format("Jan 02 2006 15:04:05")
```

**Updated Logger Code**

Here is how to integrate it into your logger so every line is stamped correctly:

**Go**

```
// Inside your logger goroutine
case s, ok := <-r.LogFileChan:
    if !ok {
        return
    }

    // Example Output: [Dec 07 2025 15:04:05] Rate: 500.00...
    timestamp := time.Now().Format("Jan 02 2006 15:04:05")

    // Prepend the timestamp to the incoming message 's'
    // Note: 's' likely already has a newline from your Sprintf calls,
    // but cleaner to ensure one exists.
    line := fmt.Sprintf("[%s] %s", timestamp, s)
    if !strings.HasSuffix(line, "\n") {
        line += "\n"
    }

    writer.WriteString(line)
    writer.Flush()
```

**User**

i have pasted my updated leader below. For some reason, many lines are missing when the worker send the completed lines to the leader. Is this a worker or a leader issue, and if it is a leader issue how can I fix it.

here is the updated leader code:

```
package main

import (
    "bufio"
    "bytes"
    "context"
    "fmt"
    "g14-mp4/RainStorm/resources"
    "g14-mp4/mp3/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "sort"
    "strconv"
    "strings"
    "sync"
    "time"
)

type WorkerIps struct {
    ips []net.IP
    sync.RWMutex
}

type RainStorm struct {
    NumStages int
    NumTasksPerStage int
    HydfsSrcDirectory string
    HydfsDestinationFileName string
    ExactlyOnce bool
    AutoScale bool
    InputRate float64
    LowestRate float64
    HighestRate float64
    Ops []Operation
    TaskInformation map[int]*TaskInfo
    NextTaskNum []int
    NextAvailableVM int
    Stage1UpdatesChan chan map[int]net.IP
    Lock sync.RWMutex
    DoneReading bool
    StartTime time.Time
   LogFile *os.File
    LogFileChan chan string
}
const clientTimeout = time.Second * 3

var workers WorkerIps
var numWorkers int
var numSuccessfulDials int
var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex
var appCompletedChan chan bool
var dataDir string

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        err := rpc.Register(&workers)
        if err != nil {
            fmt.Println(err)
            return
        }
        listener, err := net.Listen("tcp", "IntroducePort")
        if err != nil {
            fmt.Println(err)
            return
        }
        rpc.Accept(listener)()
        input := make(chan RainStorm)
        go processStdin(input)
        hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
        if err != nil {
            fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error())
            defer hydfsClient.Close()
        }
        for r := range input {
            if numWorkers == 0 {
                panic("No workers")
            }
            r.LogFileChan = make(chan string, 100)
            r.StartTime = time.Now()
            tupleListener, err := net.Listen("tcp", "TuplePort")
            if err != nil {
                return
            }
            go func() {
                path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405"))
                os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
                rLogFile, _ := os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
                rLogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Started RainStorm Application\n")
                writer := bufio.NewWriter(rLogFile)
                defer writer.Flush()
                rLogFile.Close()
            }()
            for {
                select {
                case <-ctx.Done():
                    writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
                    return
                case s, ok := <-r.LogFileChan:
                    if !ok {
                        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
                        return
                    }
                    writer.WriteString(s)
                }
            }
        }
    }()
    workers.I.RLock()
    rpcWorkers = make(map[string]*rpc.Client)
    numSuccessfulDials = 0
    rpcWorkersLock.Lock()
    for _, workerIp := range workers.ips {
        conn, err := rpc.Dial("tcp", workerIp.String() + AssignmentPort)
        if err != nil {
            fmt.Println("Unable to connect to worker: " + err.Error())
            continue
        }
        rpcWorkers[workerIp.String()] = conn
        numSuccessfulDials++
    }
    workers.I.RUnlock()
    rpcWorkersLock.Unlock()

    r.Lock = new(sync.RWMutex)
    r.Lock.Lock()
    r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
    r.NextTaskNum = make([]int, r.NumStages)
    r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
    r.DoneReading = false
    r.TaskCompletion = make([]CompletionTuple, r.NumStages)
    r.initWorker()
    r.NextAvailableVM = 0
    for i := range r.NumStages {
        r.TaskInformation[i] = make(map[int]*TaskInfo)
        r.TaskCompletion[i] = CompletionTuple{Counter: 0, StateTracker: make(map[int]bool)}
    }
    for j := range r.NumTasksPerStage {
        r.addTask(i, j)
        r.NextTaskNum[i]++
    }
    r.sendIps()
    r.Lock.Unlock()
}
```

```

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) } //buffered write to HyDFS output file outputChan := make(chan string, 200) go
func() { buffer := bytes.Buffer{} for { select { case <-ctx.Done(): if buffer.Len() > 0 { var reply
[]resources.AppendReply _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } return case line
:= <-outputChan: buffer.WriteString(line) if buffer.Len() > 4096 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } } } }()
//listen for tuples to print to console and buffered append to hydfs go func() { for { select { case <-ctx.Done():
_ = tupleListener.Close() return default: conn, err := tupleListener.Accept() if err != nil { continue } go
func(conn net.Conn) { defer conn.Close() reader := bufio.NewReader(conn) for { select { case <-ctx.Done():
return default: line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed } fmt.Print(line)
outputChan <- line } } }(conn) } }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]*WorkerClient) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Lp firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-r.Stage1UpdatesChan:
firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } numProcessed++ default: // reached if channel is empty if eofReceived {
done = true } } if done { break }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } _ = inputFile.Close() }()
// needs to
// wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan
//blocking println("RainStorm Application completed!") // CLEANUP: do once the current RainStorm application
is done cancel() close(r.LogFileChan) rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ =
worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) }
}

```

```

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] =
ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d
op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply,
string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs
when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n",
args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage
app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan
<- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] == 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages {
appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker
:= range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip:
}

```

```

workers.ips[app.NextAvailableVM%numWorkers]} workers.l.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{
TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //@TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) } }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

The issue is a combination of **Data Handling on EOF** and **Buffered Channel Draining on Shutdown**.

Here are the two specific reasons why lines are missing and how to fix them.

### Cause 1: Losing data in the Channel Buffer (The Major Leak)

You created a buffered channel: `outputChan := make(chan string, 200)`. When your application finishes and `cancel()` is called, your HyDFS writer goroutine catches `case <-ctx.Done():`. It flushes its **local** byte buffer, but it **ignores the 200 items sitting in the `outputChan`**. Those lines are instantly lost when the goroutine returns.

**The Fix:** You must drain the channel before exiting.

## Cause 2: Dropping the last line on EOF

In your listener:

```
Go Copy

line, err := reader.ReadString('\n')
if err != nil {
    return // <--- Data is lost here!
}
```

If the worker sends the last line and closes the socket simultaneously, `ReadString` returns the `line` **AND** `io.EOF` error. Your code sees the error and returns immediately, throwing away that last line.

## The Fixed Code

Here are the **two specific goroutines** you need to replace in your code to stop the data loss.

### 1. The Fixed HyDFS Writer (Drains buffer on exit)

Replace your existing `outputChan` and writer goroutine (lines 217-238) with this:

```
Go Copy

// Buffered write to HyDFS output file
outputChan := make(chan string, 200)
go func() {
    buffer := bytes.Buffer{}

    // Helper to write to HyDFS
    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }

    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            if err == io.EOF {
                flush()
                return
            }
            return
        }
        buffer.WriteString(line)
        flush()
    }
}
```

```
select {
    case <-ctx.Done():
        // 1. DRAIN THE CHANNEL: Read everything left in the pipe
        // Loop until channel is empty
        for len(outputChan) > 0 {
            line := <-outputChan
            buffer.WriteString(line)
        }
        // 2. Flush to disk
        flush()
        return
    case line := <-outputChan:
        buffer.WriteString(line)
        if buffer.Len() > 4096 {
            flush()
        }
    }
}
}()
```

## 2. The Fixed Tuple Listener (Handles EOF correctly)

Replace your existing listener goroutine (lines 241-267) with this:

Go



```
// Listen for tuples to print to console and buffered append to hydfs
go func() {
    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        fmt.Println("Error listening on tuple port:", err)
        return
    }

    // Cleanup routine to close listener on exit
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        return // Listener closed or error
    }
}
```

```
go func(c net.Conn) {
    // Sidecar to force close connection on context cancel
    // This prevents the ReadString below from hanging forever
    go func() {
        <-ctx.Done()
        c.Close()
    }()
    defer c.Close()

    reader := bufio.NewReader(c)
    for {
        line, err := reader.ReadString('\n')

        // 1. PROCESS DATA FIRST (Even if error occurred)
        if len(line) > 0 {
            fmt.Println(line)

            // Non-blocking send to avoid deadlocks during shutdown
            select {
                case outputChan <- line:
                case <-ctx.Done():
                    return
            }
        }

        // 2. CHECK ERROR AFTER PROCESSING
        if err != nil {
            return // Connection closed (EOF) or Reset
        }
    }
}(conn)
}
}()
```

**User**

ok nice, i am going to paste my updated code below again. when I am trying to run a second rainstorm application it is getting stuck somewhere:

```
package main

import (
    "bufio"
    "bytes"
    "context"
    "fmt"
    "g14-mp4/RainStorm/resources"
    "g14-mp4/mp3/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "sort"
    "strconv"
    "strings"
    "sync"
    "time"
)

type WorkerIps struct {
    ips []net.IP
    sync.RWMutex
}

type RainStorm struct {
    NumStages int
    NumTasksPerStage int
    HydfsSrcDirectory string
    HydfsDestinationFileName string
    ExactlyOnce bool
    AutoScale bool
    InputRate float64
    LowestRate float64
    HighestRate float64
    Ops []Operation
    TaskInformation []map[int]*TaskInfo
    map[int]map[int]*TaskInfo
    stage int
    task int
    IP net.IP
    PID int
    NextTaskNum []int
    NextAvailableVM int
    Stage1UpdatesChan chan map[int]net.IP
    Lock sync.RWMutex
    DoneReading bool
    StartTime time.Time
   LogFile *os.File
    LogFileChan chan string
}

const clientTimeout = time.Second * 3

var workers WorkerIps
var numWorkers int
var numSuccessfulDials int
var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex
var appCompletedChan chan bool
var dataDir string

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        err := rpc.Register(&workers)
        if err != nil {
            fmt.Println(err)
            return
        }
        listener, err := net.Listen("tcp", "IntroducePort")
        if err != nil {
            fmt.Println(err)
            return
        }
        rpc.Accept(listener)()
        input := make(chan RainStorm)
        go processStdin(input)
        hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
        if err != nil {
            fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error())
            defer hydfsClient.Close()
        }
        for r := range input {
            if numWorkers == 0 {
                panic("No workers")
            }
            var wg sync.WaitGroup
            r.LogFileChan = make(chan string, 100)
            r.StartTime = time.Now()
            tupleListener, err := net.Listen("tcp", "TuplePort")
            if err != nil {
                continue
            }
            go func() {
                path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405"))
                os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
                r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
                _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Started RainStorm Application\n")
                writer := bufio.NewWriter(r.LogFile)
                defer writer.Flush()
                r.LogFile.Close()
            }()
            for {
                select {
                case <-ctx.Done():
                    writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
                    return
                case s, ok := <-r.LogFileChan:
                    if !ok {
                        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
                        return
                    }
                    writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s)
                    writer.Flush()
                }
            }()
            // INITIATE NEW RAINSTORM APPLICATION //Global RM /* 1.
            open listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3. compare rates to see if changes are needed 4. complete changes */
            appServer := rpc.NewServer()
            err = appServer.Register(&r)
            if err != nil {
                fmt.Println(err)
                continue
            }
            globalRmListener, err := net.Listen("tcp", "GlobalRMPort")
            if err != nil {
                fmt.Println("GlobalRM unable to connect to worker: " + err.Error())
                continue
            }
            go func() {
                for {
                    conn, err := globalRmListener.Accept()
                    if err != nil {
                        if strings.Contains(err.Error(), "use of closed network connection") {
                            return
                        }
                    }
                    // Exit quietly
                    fmt.Println("GlobalRM Accept error: " + err.Error())
                    return
                }
            }()
            //give connection to RPC server
            go appServer.ServeConn(conn)
        }
        workers.I.RLock()
        rpcWorkers = make(map[string]*rpc.Client)
        numSuccessfulDials = 0
        rpcWorkersLock.Lock()
        for _, workerIp := range workers.ips {
            //collect list of tasks for this worker
            err := rpc.Dial("tcp", workerIp.String() + AssignmentPort)
            if err != nil {
                fmt.Println("Unable to connect to worker: " + err.Error())
                continue
            }
            rpcWorkers[workerIp.String()] = worker
            numSuccessfulDials++
        }
        workers.I.RUnlock()
        rpcWorkersLock.Unlock()
        r.Lock = new(sync.RWMutex)
        r.Lock.Lock()
        r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
        r.NextTaskNum = make([]int, r.NumStages)
        r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
        r.DoneReading = false
        r.TaskCompletion = make([]CompletionTuple, r.NumStages)
        r.initWorker()
        r.NextAvailableVM = 0
        for i := range r.NumStages {
            r.TaskInformation[i] = make(map[int]*TaskInfo)
            r.TaskCompletion[i] = CompletionTuple{Counter: 0, StateTracker: make(map[int]bool)}
        }
        for j := range r.NumTasksPerStage {
            r.addTask(i, j)
            r.NextTaskNum[i]++
        }
        r.sendIps()
        r.Lock.Unlock()
    }
}
```

```

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) } //buffered write to HyDFS output file outputChan := make(chan string, 200) go
func() { buffer := bytes.Buffer{} flush := func() { if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } } for { select { case <-
ctx.Done(): for len(outputChan) > 0 { line := <-outputChan buffer.WriteString(line) } if buffer.Len() > 0 { flush()
} return case line := <-outputChan: buffer.WriteString(line) if buffer.Len() > 4096 { flush() } } } }()
//listen for tuples to print to console and buffered append to hydfs go func() { for { select { case <-ctx.Done():
return default: conn, err := tupleListener.Accept() if err != nil { continue } wg.Add(1) go func(conn net.Conn) {
defer wg.Done() defer conn.Close() reader := bufio.NewReader(conn) for { select { case <-ctx.Done(): return
default: line, err := reader.ReadString('\n') if err != nil { return // connection closed/failed } fmt.Print(line)
outputChan <- line } } }(conn) } } }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }() eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _,
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } numProcessed++ default: // reached if channel is empty if eofReceived {
done = true } } if done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } _ = inputFile.Close() }()
// needs to
wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan
//blocking println("RainStorm Application completed!") // CLEANUP: do once the current RainStorm application
is done wg.Wait() cancel() close(r.LogFileChan) _ = tupleListener.Close() rpcWorkersLock.Lock() for _, worker
:= range rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) }
}
func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()

```

```

workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.I.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.I.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] =
ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d
op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply,
string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { // @TODO: write to leader logs
when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n",
args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage
app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan
<- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages {
appCompletedChan <- true } }

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stagID to workers: " + x.Error.Error()) } } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker
:= range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }

func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.I.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip:
workers.ips[app.NextAvailableVM%numWorkers]} workers.I.RUnlock()
}

```

```
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip } app.Stage1UpdatesChan <- temp } task := Task{
    TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name))
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoScaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil {
fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: // AutoScale
rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate
rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //TODO: add implementation for this break
case "list_tasks": break
} } }
```

## Gemini

This is a complex concurrency problem. You are experiencing **race conditions on shutdown**.

Here is exactly why your second app gets stuck and why you are losing data:

- 1. Data Loss (The "Race to Exit"):** When `appCompletedChan` triggers, you call `cancel()`. The HyDFS Writer goroutine sees `ctx.Done()` and exits immediately. However, the `TupleListener` might still have

data buffered that it hasn't sent yet, or `outputChan` might still have data in it. You are killing the writer before it drains the queue.

2. **Stuck on Second App (Port Conflicts):** When the loop restarts, it tries to `net.Listen` on `GlobalRMPort` and `TuplePort` immediately. If the Operating System hasn't fully released these ports (which takes a few milliseconds), `net.Listen` fails. Your code catches the error, prints it, and **continues to the top of the loop**. This means it goes back to `r := <-input` and waits for you to type the command again. It *looks* stuck, but it's actually waiting for input because the startup failed.

## The Fix: "Graceful Shutdown Pipeline"

You need a strict order of operations for shutdown to ensure no data is lost and ports are free:

1. **Cancel Context** (Tells Listener to stop accepting).
2. **Wait for Listener to Exit** (Ensures it stops writing to channel).
3. **Close Output Channel** (Tells Writer "no more data coming").
4. **Wait for Writer to Finish** (Ensures all data is flushed to HyDFS).
5. **Close Ports**.

Here is the corrected code.

## Changes Made:

1. **Added `writerDone` channel:** To wait for HyDFS flushing to finish.
2. **Changed Writer Logic:** It now iterates `range outputChan`. It only stops when the channel is closed, guaranteeing it drains everything.
3. **Retry Loop for Ports:** Added a retry loop for `net.Listen`. If the port is busy from the previous run, it waits 1 second and tries again, instead of failing silently.
4. **WaitGroups:** Properly utilized `wg` to ensure the listener is fully dead before closing the channel.

Go



```
package main

import (
    "bufio"
    "bytes"
    "context"
    "fmt"
    ". \"g14-mp4/RainStorm/resources\""
    "g14-mp4/mp3/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "sort"
    "strconv"
    "strings"
    "sync"
```

```
"time"  
)  
  
type WorkerIps struct {  
    ips []net.IP  
    l   sync.RWMutex  
}  
  
type RainStorm struct {  
    NumStages           int  
    NumTasksPerStage   int  
    HydfsSrcDirectory string  
    HydfsDestinationFileName string  
    ExactlyOnce         bool  
    AutoScale           bool  
    InputRate            float64  
    LowestRate           float64  
    HighestRate          float64  
    Ops                 []Operation  
    TaskInformation     []map[int]*TaskInfo  
    NextTaskNum          []int  
    NextAvailableVM      int  
    Stage1UpdatesChan   chan map[int]net.IP  
    Lock                *sync.RWMutex  
    DoneReading          bool  
    StartTime            time.Time  
   LogFile              *os.File  
    LogFileChan          chan string  
}  
  
const clientTimeout = time.Second * 3  
  
var workers WorkerIps  
var numWorkers int  
var rpcWorkers map[string]*rpc.Client  
var rpcWorkersLock sync.RWMutex  
var appCompletedChan chan bool  
var dataDir string  
  
func main() {  
    homeDir, _ := os.UserHomeDir()  
    dataDir = filepath.Join(homeDir, "data")  
    workers = WorkerIps{  
        appCompletedChan: make(chan bool, 1)  
    }  
  
    // Register generic workers RPC once outside the loop
```

```
go func() {
    err := rpc.Register(&workers)
    if err != nil {
        fmt.Println(err)
        return
    }
    listener, err := net.Listen("tcp", IntroducePort)
    if err != nil {
        fmt.Println(err)
        return
    }
    rpc.Accept(listener)
}()

input := make(chan RainStorm)
go processStdin(input)

hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
if err != nil {
    fmt.Println("Failed to connect to HyDFS client: " + err.Error())
}
defer hydfsClient.Close()

for {
    r := <-input
    if numWorkers == 0 {
        panic("No workers available")
    }

    // RESET STATE FOR NEW APP
    ctx, cancel := context.WithCancel(context.Background())
    var wg sync.WaitGroup // Waits for Listener to exit

    r.LogFileChan = make(chan string, 500)
    r.StartTime = time.Now()

    // --- RETRY LOOP FOR TUPLE PORT (Fixes "Stuck" Issue) ---
    var tupleListener net.Listener
    for i := 0; i < 5; i++ {
        tupleListener, err = net.Listen("tcp", TuplePort)
        if err == nil {
            break
        }
        fmt.Println("TuplePort busy, retrying in 1s...")
        time.Sleep(1 * time.Second)
    }
}
```

```
if err != nil {
    fmt.Println("Failed to bind TuplePort, aborting app:", err)
    cancel()
    continue
}

// --- LOGGER ---
// Use a wait group or done channel for logger too
logDone := make(chan bool)
go func() {
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("")
    _ = os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
    _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Start

    writer := bufio.NewWriter(r.LogFile)
    defer func() {
        writer.Flush()
        r.LogFile.Close()
        close(logDone)
    }()
}

for {
    select {
    case <-ctx.Done():
        // Drain remaining logs
        for len(r.LogFileChan) > 0 {
            s := <-r.LogFileChan
            writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
        }
        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm"
        return
    case s, ok := <-r.LogFileChan:
        if !ok { return }
        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
        writer.Flush()
    }
}
}()

// --- GLOBAL RM SETUP ---
appServer := rpc.NewServer()
err = appServer.Register(&r)
if err != nil {
    fmt.Println(err)
    cancel()
```

```
        continue
    }

    // Retry loop for GlobalRM port
    var globalRmListener net.Listener
    for i := 0; i < 5; i++ {
        globalRmListener, err = net.Listen("tcp", GlobalRMPort)
        if err == nil {
            break
        }
        time.Sleep(1 * time.Second)
    }
    if err != nil {
        fmt.Println("Failed to bind GlobalRM port")
        cancel()
        continue
    }

    go func() {
        for {
            conn, err := globalRmListener.Accept()
            if err != nil {
                return
            }
            go appServer.ServeConn(conn)
        }
    }()
}

// --- DIAL WORKERS ---
workers.l.RLock()
rpcWorkers = make(map[string]*rpc.Client)
rpcWorkersLock.Lock()
for _, workerIp := range workers.ips {
    worker, err := rpc.Dial("tcp", workerIp.String()+AssignmentPort)
    if err != nil {
        fmt.Println("Unable to connect to worker: " + err.Error())
        continue
    }
    rpcWorkers[workerIp.String()] = worker
}
workers.l.RUnlock()
rpcWorkersLock.Unlock()

// --- INITIALIZE STRUCTS ---
r.Lock = new(sync.RWMutex)
r.Lock.Lock()
```

```
r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages)
r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false
r.initWorker()
r.NextAvailableVM = 0
for i := range r.NumStages {
    r.TaskInformation[i] = make(map[int]*TaskInfo)
    for j := range r.NumTasksPerStage {
        r.addTask(i, j)
        r.NextTaskNum[i]++
    }
}
r.sendIps()
r.Lock.Unlock()

// --- HYDFS FILE CREATION ---
var createReply []resources.AddFileReply
_ = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
    RemoteName: r.HydfsDestinationFileName,
    Content:    make([]byte, 0),
}, &createReply)

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))
if err != nil {
    fmt.Println("Unable to open src directory: " + err.Error())
}

// --- HYDFS WRITER ---
outputChan := make(chan string, 500)
writerDone := make(chan bool) // Signal when writing is totally done

go func() {
    buffer := bytes.Buffer{}
    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
}

// Range loop automatically waits until channel is closed AND empty
```

```
for line := range outputChan {
    buffer.WriteString(line)
    if buffer.Len() > 4096 {
        flush()
    }
}
// Channel is closed, flush remaining
flush()
close(writerDone)
}()

// --- TUPLE LISTENER ---
wg.Add(1)
go func() {
    defer wg.Done()
    // Cleanup routine
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        return
    }

    wg.Add(1) // Wait for individual connections too
    go func(conn net.Conn) {
        defer wg.Done()
        defer conn.Close()

        // Sidecar closer
        go func() {
            <-ctx.Done()
            conn.Close()
        }()
    }

    reader := bufio.NewReader(conn)
    for {
        line, err := reader.ReadString('\n')
        if err != nil {
            return
        }
        fmt.Println(line) // Console feedback
    }
}
```

```
// Non-blocking send to avoid hanging if writer dies
select {
    case outputChan <- line:
    case <-ctx.Done():
        return
    }
}
}(conn)
}
}()

// --- INPUT READER (SOURCE) ---
go func() {
    scanner := bufio.NewScanner(inputFile)

    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0)
    for tid, info := range r.TaskInformation[0] {
        firstStageTasks[tid] = info.Ip
        firstTaskList = append(firstTaskList, tid)
    }
    sort.Ints(firstTaskList)
    r.Lock.Unlock()

    tupleClients := make(map[string]*WorkerClient, numWorkers)
    startTime := time.Now()
    var numProcessed float64 = 0

    readingChan := make(chan struct {
        line    string
        lineNum int
    }, 100)

    // Producer
    go func() {
        lineNum := 0
        for scanner.Scan() {
            readingChan <- struct {
                line    string
                lineNum int
            }{line: scanner.Text(), lineNum: lineNum}
            lineNum++
        }
        readingChan <- struct {
            line    string
        }
    }()
}
```

```
    lineNumber int
    }{line: "", lineNumber: -1}

    r.Lock.Lock()
    r.DoneReading = true
    r.Lock.Unlock()
}()

// Consumer
for {
    select {
    case tuple := <-readingChan:
        if tuple.lineNum == -1 {
            goto EndOfFile
        }

        // Check for updates
        select {
        case updatedMap := <-r.Stage1UpdatesChan:
            firstStageTasks = updatedMap
            firstTaskList = make([]int, 0)
            for k := range firstStageTasks {
                firstTaskList = append(firstTaskList, k)
            }
            sort.Ints(firstTaskList)
        default:
        }

        // FIX: Divide by zero check
        if len(firstTaskList) == 0 {
            time.Sleep(50 * time.Millisecond)
            readingChan <- tuple
            continue
        }

        nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
        nextTaskIp := firstStageTasks[nextTask]

        client, ok := tupleClients[nextTaskIp.String()]
        if !ok {
            conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
            if err != nil {
                fmt.Println("Unable to connect to worker: " + err.Error())
                delete(tupleClients, nextTaskIp.String())
                continue
            }
        }
    }
}
```

```
        client = &WorkerClient{
            Conn: conn,
            Buf:  bufio.NewReader(conn),
        }
        tupleClients[nextTaskIp.String()] = client
    }

    _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
_, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%s\n", "temp", tuple.lineNum,

    _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

    if err != nil || strings.TrimSpace(ack) != expectedAck {
        client.Conn.Close()
        delete(tupleClients, nextTaskIp.String())
        readingChan <- tuple
        continue
    }

expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
targetTime := startTime.Add(expectedDuration)
now := time.Now()
if targetTime.After(now) {
    time.Sleep(targetTime.Sub(now))
}
numProcessed++
}

}

EndOfFile:
r.sendStageCompletion(-1)
for _, c := range tupleClients {
    c.Conn.Close()
}
_ = inputFile.Close()
}()

// --- WAIT FOR APP COMPLETION ---
<-appCompletedChan // Block until done
println("RainStorm Application completed! Cleaning up...")

// --- SHUTDOWN SEQUENCE (Order Matters!) ---
// 1. Cancel context (Stops Input, Stops Tuple Listener accepting, Stops Log)
cancel()
```

```
// 2. Wait for Tuple Listener and all its connections to finish reading/writing
wg.Wait()

// 3. Close the Output Channel (Signals HyDFS writer that no more data exists)
close(outputChan)

// 4. Wait for HyDFS Writer to finish flushing to disk
<-writerDone

// 5. Cleanup Resources
<-logDone // Wait for logger to finish

rpcWorkersLock.Lock()
for _, worker := range rpcWorkers {
    _ = worker.Close()
}
rpcWorkersLock.Unlock()

err = globalRmListener.Close()
if err != nil {
    fmt.Println(err)
}

println("Cleanup finished. Ready for next app.")
}

}

// ... (Rest of your methods: AddWorker, ReceiveFailure, etc. keep them as is with the sele
```

**User**

the ports being released is not an issue

**Gemini**

If port conflicts aren't the issue, then your application is getting stuck because of a **deadlock in your Tuple Listener shutdown logic**.

## The Deadlock Explained

In your listener goroutine, you have this structure:

```
Go □  
  
select {  
    case <-ctx.Done():  
        _ = tupleListener.Close() // <--- This never runs!  
        return  
    default:  
        conn, err := tupleListener.Accept() // <--- BLOCKING CALL  
    }  
  
}
```

**Why it hangs:** `Accept()` is a blocking call. If no workers are trying to connect, the code stops at that line forever. It **cannot** go back up to check `case <-ctx.Done()`. Since the listener is never closed, `Accept` never errors out, and the goroutine never exits.

## The Fix

You must close the listener from a **separate goroutine** (or the main thread). This forces `Accept()` to return an error immediately, breaking the deadlock.

Here is the corrected, simplified code. I have removed the port retry logic as requested and fixed the listener deadlock.

```
Go □  
  
package main  
  
import (  
    "bufio"  
    "bytes"  
    "context"  
    "fmt"  
    . "g14-mp4/RainStorm/resources"  
    "g14-mp4/mp3/resources"  
    "net"  
    "net/rpc"  
    "os"  
    "path/filepath"  
)
```

```
"sort"
"strconv"
"strings"
"sync"
"time"
)

// ... [Structs WorkerIps and RainStorm remain unchanged] ...

// ... [Global variables remain unchanged] ...

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)

    // RPC Register (Run once)
    go func() {
        _ = rpc.Register(&workers)
        listener, _ := net.Listen("tcp", IntroducePort)
        rpc.Accept(listener)
    }()

    input := make(chan RainStorm)
    go processStdin(input)
    hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
    if err != nil {
        fmt.Println("Failed to connect to HyDFS:", err)
    }
    defer hydfsClient.Close()

    for {
        r := <-input
        if numWorkers == 0 {
            panic("No workers")
        }

        // Context for cancellation
        ctx, cancel := context.WithCancel(context.Background())

        // WaitGroup to ensure clean shutdown of background routines
        var wg sync.WaitGroup

        r.LogFileChan = make(chan string, 100)
        r.StartTime = time.Now()
```

```
// Setup Listeners
tupleListener, err := net.Listen("tcp", TuplePort)
if err != nil {
    fmt.Println("TuplePort Error:", err)
    continue
}

// --- LOGGER ---
wg.Add(1)
go func() {
    defer wg.Done()
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("")
    _ = os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
    _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Start

    writer := bufio.NewWriter(r.LogFile)
    defer func() {
        writer.Flush()
        r.LogFile.Close()
    }()
}

for {
    select {
    case <-ctx.Done():
        // Drain channel then exit
        for len(r.LogFileChan) > 0 {
            s := <-r.LogFileChan
            writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
        }
        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm")
        return
    case s, ok := <-r.LogFileChan:
        if !ok { return }
        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
        writer.Flush()
    }
}
}()

// --- GLOBAL RM ---
appServer := rpc.NewServer()
_ = appServer.Register(&r)
globalRmListener, err := net.Listen("tcp", GlobalRMPort)
if err != nil {
```

```
        fmt.Println("GlobalRM Port Error:", err)
        continue
    }

wg.Add(1)
go func() {
    defer wg.Done()
    // Force close listener on context cancel to unblock Accept()
    go func() { <-ctx.Done(); globalRmListener.Close() }()

    for {
        conn, err := globalRmListener.Accept()
        if err != nil {
            return // Exit loop when listener is closed
        }
        go appServer.ServeConn(conn)
    }
}()

// ... [Dialing Workers logic remains unchanged] ...

// ... [Initialize Structs logic remains unchanged] ...

// ... [HyDFS File Create logic remains unchanged] ...

// --- HYDFS BUFFERED WRITER ---
outputChan := make(chan string, 200)
wg.Add(1)
go func() {
    defer wg.Done()
    buffer := bytes.Buffer{}

    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
}

for {
    select {
    case <-ctx.Done():
        flush()
        wg.Wait()
        return
    }
}
```

```
// Drain remaining data in channel
for len(outputChan) > 0 {
    line := <-outputChan
    buffer.WriteString(line)
}
flush()
return

case line := <-outputChan:
    buffer.WriteString(line)
    if buffer.Len() > 4096 {
        flush()
    }
}

}()

// --- TUPLE LISTENER (FIXED DEADLOCK) ---
wg.Add(1)
go func() {
    defer wg.Done()

    // 1. Sideline Closer: Waits for context, closes listener to unblock Accept
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    // 2. Accept blocks here. When Close() is called above, this returns error imm
    conn, err := tupleListener.Accept()
    if err != nil {
        return
    }

    // Handle connection
    wg.Add(1)
    go func(c net.Conn) {
        defer wg.Done()
        defer c.Close()

        // Connection specific closer
        go func() { <-ctx.Done(); c.Close() }()

        reader := bufio.NewReader(c)
        for {
            line, err := reader.ReadString('\n')

```

```
        if err != nil {
            return
        }
        fmt.Println(line)

        select {
        case outputChan <- line:
        case <-ctx.Done():
            return
        }
    }
}(conn)
}

// --- INPUT READER ---
go func() {
    // ... [Scanner setup remains unchanged] ...

    // ... [Producer loop remains unchanged] ...

    // Consumer
    for {
        select {
        case tuple := <-readingChan:
            if tuple.lineNum == -1 {
                goto EndOfFile
            }
            // ... [Topology update select logic remains unchanged] ...

            // FIX: Divide by zero check
            if len(firstTaskList) == 0 {
                time.Sleep(50 * time.Millisecond)
                readingChan <- tuple
                continue
            }

            // ... [Dialing and Sending logic remains unchanged] ...

            // ... [Rate Limiting logic remains unchanged] ...
        }
    }

    EndOfFile:
    r.sendStageCompletion(-1)
    for _, c := range tupleClients {
```

```
        c.Conn.Close()
    }
    _ = inputFile.Close()
}()

// --- CLEANUP ---
<-appCompletedChan
println("RainStorm Application completed!")

// 1. Trigger shutdown
cancel()

// 2. Wait for all listeners, loggers, and writers to finish
// This ensures HyDFS writer flushes and Listener loop breaks
wg.Wait()

rpcWorkersLock.Lock()
for _, worker := range rpcWorkers {
    _ = worker.Close()
}
rpcWorkersLock.Unlock()
}

}
```

**User**

does this fix it:

```
package main

import ("bufio" "bytes" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net"
"net/rpc" "os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }

type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.FileLogFileChan chan string }
const clientTimeout = time.Second * 3

var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1) ctx, cancel := context.WithCancel(context.Background())
go func() { err := rpc.Register(&workers) if err != nil { fmt.Println(err) return } listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println(err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go
processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS client if
err != nil { fmt.Println("Failed to connect to HyDFS client from leader: " + err.Error()) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } var wg sync.WaitGroup
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now()
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { continue } wg.Add(1) go func() { defer wg.Done()
path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755) r.LogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02
15:04:05") + ": Started RainStorm Application\n")
writer := bufio.NewWriter(r.LogFile) defer func() { writer.Flush() r.LogFile.Close() }()
for { select { case <-ctx.Done(): // Drain channel then exit for len(r.LogFileChan) > 0 { s := <-r.LogFileChan
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s)
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
return case s, ok := <-r.LogFileChan: if !ok { return } writer.WriteString(time.Now().Format("2006-01-02
15:04:05") + ": " + s) writer.Flush() } } }()
INITIATE NEW RAINSTORM APPLICATION //Global RM /* 1. open
listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3. compare rates to see if
changes are needed 4. complete changes */ appServer := rpc.NewServer() err = appServer.Register(&r) if err
!= nil { fmt.Println(err) continue } globalRmListener, err := net.Listen("tcp", GlobalRMPort) if err != nil {
fmt.Println("GlobalRM unable to connect to worker: " + err.Error()) continue } go func() { for { conn, err :=
globalRmListener.Accept() if err != nil { if strings.Contains(err.Error(), "use of closed network connection") {
return // Exit quietly } fmt.Println("GlobalRM Accept error: " + err.Error()) return } //give connection to RPC
server go appServer.ServeConn(conn) }()
workers.l.RLock() rpcWorkers = make(map[string]*rpc.Client) numSuccessfulDials = 0 rpcWorkersLock.Lock()
for _, workerIp := range workers.ips { //collect list of tasks for this worker worker, err := rpc.Dial("tcp",
workerIp.String() + AssignmentPort) if err != nil { fmt.Println("Unable to connect to worker: " + err.Error())
continue } rpcWorkers[workerIp.String()] = worker numSuccessfulDials++ } workers.l.RUnlock()
rpcWorkersLock.Unlock()

r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false //r.TaskCompletion = make([]CompletionTuple, r.NumStages) r.initWorker()
r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] = make(map[int]*TaskInfo)
//r.TaskCompletion[i] = CompletionTuple{ // Counter: 0, // StateTracker: make(map[int]bool), //} for j := range
r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps() r.Lock.Unlock()
```

```

//@TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
RemoteName: r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) if err != nil {
fmt.Println(err.Error()) }
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) } //buffered write to HyDFS output file outputChan := make(chan string, 200)
wg.Add(1) go func() { defer wg.Done() buffer := bytes.Buffer{}
flush := func() { if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } }
for { select { case <-ctx.Done(): // Drain remaining data in channel for len(outputChan) > 0 { line := <-
outputChan buffer.WriteString(line) } flush() return case line := <-outputChan: buffer.WriteString(line) if
buffer.Len() > 4096 { flush() } } } }()
// --- TUPLE LISTENER (FIXED DEADLOCK) --- wg.Add(1) go func() { defer wg.Done()
// 1. Sideline Closer: Waits for context, closes listener to unblock Accept go func() { <-ctx.Done()
tupleListener.Close() }()
for { // 2. Accept blocks here. When Close() is called above, this returns error immediately. conn, err :=
tupleListener.Accept() if err != nil { return }
// Handle connection wg.Add(1) go func(c net.Conn) { defer wg.Done() defer c.Close()
// Connection specific closer go func() { <-ctx.Done(); c.Close() }()
reader := bufio.NewReader(c) for { line, err := reader.ReadString('\n') if err != nil { return } fmt.Print(line)
select { case outputChan <- line: case <-ctx.Done(): return } } }(conn) }()
// reading src file and sending lines to tasks go func() { scanner := bufio.NewScanner(inputFile) r.Lock.Lock()
firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range r.TaskInformation[0]
{ firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) } sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) startTime := time.Now() var numProcessed
float64 = 0 readingChan := make(chan struct { line string lineNumber int }, 100) go func() { lineNumber := 0 for
scanner.Scan() { readingChan <- struct { line string lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading
= true r.Lock.Unlock() }()
eofReceived := false done := false for { select { case tuple := <-readingChan: if
tuple.lineNumber == -1 { //done reading eofReceived = true continue } select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range
firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Unable to connect to worker: " + err.Error()) delete(tupleClients, nextTaskIp.String())
continue } client = &WorkerClient{ Conn: conn, Buf: bufio.NewReader(conn), } tupleClients[nextTaskIp.String()] =
client }
// Send the tuple _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) // Id-Id, stage, task, data _,
_ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
// Wait for the ack _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK") if err != nil
|| strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients, nextTaskIp.String())
readingChan <- tuple continue }
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second)) targetTime :=
startTime.Add(expectedDuration)
now := time.Now() if targetTime.After(now) { // ahead of schedule, sleep to sync with desired rate
time.Sleep(targetTime.Sub(now)) } numProcessed++ default: // reached if channel is empty if eofReceived {
done = true } } if done { break } }
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() _ = inputFile.Close() }()
// needs to
wait for the application to complete before cleaning up --> @TODO: come back to this <-appCompletedChan

```

```
//blocking println("RainStorm Application completed!") // CLEANUP: do once the current RainStorm application
is done wg.Wait() cancel() close(r.LogFileChan) _ = tupleListener.Close() rpcWorkersLock.Lock() for _, worker
:= range rpcWorkers { _ = worker.Close() } rpcWorkersLock.Unlock()
err = globalRmListener.Close() if err != nil { fmt.Println(err) } }

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d
op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply,
string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //@TODO: write to leader logs
when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n",
args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage
app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan
<- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages {
appCompletedChan <- true } }

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker
:= range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
```

```

func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.I.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip:
workers.ips[app.NextAvailableVM%numWorkers]} workers.I.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip :=
range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } task := Task{
TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err
!= nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)) }

func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp :=
make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip }
app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err :=
rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }

func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin :=
scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //TODO: add implementation for this break
case "list_tasks": break
} } }

```

## Gemini

The primary reason your second application gets stuck is a **deadlock in the TupleListener shutdown logic**.

When your first app finishes, you call `cancel()`, but the `TupleListener` loop inside `go func()` gets stuck on `conn, err := tupleListener.Accept()`. Because `Accept()` blocks forever until a connection arrives or the listener is closed, and you only close the listener *after* the loop exits (which it never does), the goroutine never finishes. The `wg.Wait()` in the main thread then blocks forever, preventing the second app from starting.

Here is the fixed, production-ready `main` function. I have fixed the deadlock, the data loss race condition, and the port cleanup order.

## The Fix

Replace your entire `main` function with this.

Go



```
func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)

    // 1. Start the RPC Server ONCE (outside the loop)
    // We do not want to re-register "WorkerIps" every time causing "multiple registration"
    go func() {
        _ = rpc.Register(&workers)
        listener, err := net.Listen("tcp", IntroducePort)
        if err != nil {
            fmt.Println("IntroducePort Error:", err)
            return
        }
        rpc.Accept(listener)
    }()

    input := make(chan RainStorm)
    go processStdin(input)

    // Connect to HyDFS once
    hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
    if err != nil {
        fmt.Println("Failed to connect to HyDFS client:", err)
    }
    defer hydfsClient.Close()

    // --- MAIN APPLICATION LOOP ---
    for {
```

```
r := <-input
if numWorkers == 0 {
    panic("No workers available")
}

// Reset State
ctx, cancel := context.WithCancel(context.Background())
var wg sync.WaitGroup

r.LogFileChan = make(chan string, 500)
r.StartTime = time.Now()

// --- 1. SETUP LISTENERS ---
tupleListener, err := net.Listen("tcp", TuplePort)
if err != nil {
    fmt.Println("TuplePort bind error:", err)
    cancel()
    continue
}

globalRmListener, err := net.Listen("tcp", GlobalRMPort)
if err != nil {
    fmt.Println("GlobalRM bind error:", err)
    _ = tupleListener.Close()
    cancel()
    continue
}

// --- 2. START LOGGER ---
wg.Add(1)
go func() {
    defer wg.Done()
    path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format(""))
    _ = os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)

    r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
    _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Start

    writer := bufio.NewWriter(r.LogFile)
    defer func() {
        writer.Flush()
        r.LogFile.Close()
    }()
}

for {
    select {
```

```
        case <-ctx.Done():
            // Drain remaining logs
            for len(r.LogFileChan) > 0 {
                s := <-r.LogFileChan
                writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s)
            }
            writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm"
            return
        case s, ok := <-r.LogFileChan:
            if !ok { return }
            writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s)
            writer.Flush()
        }
    }
}()
```

// --- 3. START GLOBAL RM ---

```
appServer := rpc.NewServer()
_ = appServer.Register(&r)

wg.Add(1)
go func() {
    defer wg.Done()
    // Sidecar to close listener on exit, breaking the Accept loop
    go func() { <-ctx.Done(); globalRmListener.Close() }()
}

for {
    conn, err := globalRmListener.Accept()
    if err != nil {
        return // Listener closed
    }
    go appServer.ServeConn(conn)
}
}()
```

// --- 4. CONNECT TO WORKERS ---

```
workers.l.RLock()
rpcWorkers = make(map[string]*rpc.Client)
rpcWorkersLock.Lock()
for _, workerIp := range workers.ips {
    worker, err := rpc.Dial("tcp", workerIp.String()+AssignmentPort)
    if err != nil {
        continue
    }
    rpcWorkers[workerIp.String()] = worker
}
```

```
workers.l.RUnlock()
rpcWorkersLock.Unlock()

// --- 5. INITIALIZE TOPOLOGY ---
r.Lock = new(sync.RWMutex)
r.Lock.Lock()
r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages)
r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false
r.initWorker()
r.NextAvailableVM = 0
for i := range r.NumStages {
    r.TaskInformation[i] = make(map[int]*TaskInfo)
    for j := range r.NumTasksPerStage {
        r.addTask(i, j)
        r.NextTaskNum[i]++
    }
}
r.sendIps()
r.Lock.Unlock()

// --- 6. START HYDFS OUTPUT WRITER ---
// Initialize HyDFS file
var createReply []resources.AddFileReply
_ = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
    RemoteName: r.HydfsDestinationFileName,
    Content:    make([]byte, 0),
}, &createReply)

outputChan := make(chan string, 500)

wg.Add(1)
go func() {
    defer wg.Done()
    buffer := bytes.Buffer{}

    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
}
```

```
        }

    for {
        select {
        case <-ctx.Done():
            // DRAIN: Write everything remaining in channel
            for len(outputChan) > 0 {
                line := <-outputChan
                buffer.WriteString(line)
            }
            flush()
            return
        case line := <-outputChan:
            buffer.WriteString(line)
            if buffer.Len() > 4096 {
                flush()
            }
        }
    }
}()
```

// --- 7. START TUPLE LISTENER (Output Collector) ---

```
wg.Add(1)
go func() {
    defer wg.Done()

    // Sidecar: Close listener when context dies
    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        return
    }

    wg.Add(1) // Wait for individual connections
    go func(c net.Conn) {
        defer wg.Done()
        defer c.Close()

        // Connection Sidecar: Force close to unblock ReadString
        go func() { <-ctx.Done(); c.Close() }()
    }()
}
```

```
reader := bufio.NewReader(c)
for {
    line, err := reader.ReadString('\n')
    if len(line) > 0 {
        fmt.Println(line)
        // Non-blocking send
        select {
            case outputChan <- line:
            case <-ctx.Done():
                return
        }
    }
    if err != nil {
        return
    }
}
}(conn)
}

// --- 8. START INPUT READER (Source) ---
inputFile, _ := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))

wg.Add(1)
go func() {
    defer wg.Done()
    defer inputFile.Close()

    scanner := bufio.NewScanner(inputFile)

    // ... [Your existing Snapshot Logic Here] ...
    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0)
    for tid, info := range r.TaskInformation[0] {
        firstStageTasks[tid] = info.Ip
        firstTaskList = append(firstTaskList, tid)
    }
    sort.Ints(firstTaskList)
    r.Lock.Unlock()

    tupleClients := make(map[string]*WorkerClient, numWorkers)
    var numProcessed float64 = 0
    startTime := time.Now() // Local start time for rate limiting

    // Reading Channel
```

```
readingChan := make(chan struct{ line string; lineNumber int }, 100)

// Producer Routine
go func() {
    lineNumber := 0
    for scanner.Scan() {
        readingChan <- struct{ line string; lineNumber int }{line: scanner.Text(), lineNumber}
        lineNumber++
    }
    readingChan <- struct{ line string; lineNumber int }{line: "", lineNumber: -1}

    r.Lock.Lock()
    r.DoneReading = true
    r.Lock.Unlock()
}()

// Consumer Routine
for {
    select {
    case <-ctx.Done():
        // Cleanup clients
        for _, c := range tupleClients { c.Conn.Close() }
        return

    case tuple := <-readingChan:
        if tuple.lineNumber == -1 {
            r.sendStageCompletion(-1)
            for _, c := range tupleClients { c.Conn.Close() }
            return // Done reading
        }

        // ... [Update Logic] ...
        select {
        case updatedMap := <-r.Stage1UpdatesChan:
            firstStageTasks = updatedMap
            firstTaskList = make([]int, 0)
            for k := range firstStageTasks {
                firstTaskList = append(firstTaskList, k)
            }
            sort.Ints(firstTaskList)
        default:
        }

        // ... [Panic Fix] ...
        if len(firstTaskList) == 0 {
            time.Sleep(50 * time.Millisecond)
        }
    }
}
```

```
        readingChan <- tuple
        continue
    }

    // ... [Routing Logic] ...
    nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
    nextTaskIp := firstStageTasks[nextTask]

    client, ok := tupleClients[nextTaskIp.String()]
    if !ok {
        conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
        if err != nil {
            // Retry logic
            delete(tupleClients, nextTaskIp.String())
            continue
        }
        client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
        tupleClients[nextTaskIp.String()] = client
    }

    // ... [Sending Logic] ...
    _, _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
    _, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%s\n", "temp", tuple.lineNum,
                       tuple.lineNum)

    // Wait ACK
    _, _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
    ack, err := client.Buf.ReadString('\n')
    expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

    if err != nil || strings.TrimSpace(ack) != expectedAck {
        client.Conn.Close()
        delete(tupleClients, nextTaskIp.String())
        readingChan <- tuple
        continue
    }

    // Rate Limit
    numProcessed++
    expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
    targetTime := startTime.Add(expectedDuration)
    if targetTime.After(time.Now()) {
        time.Sleep(targetTime.Sub(time.Now())))
    }
}

}()
```

```
// --- 9. WAIT FOR COMPLETION AND CLEANUP ---
<-appCompletedChan
fmt.Println("App completed. Shutting down....")

// Stop all background routines
cancel()

// Wait for Listeners, Writers, Logger to finish
wg.Wait()

// Close workers
rpcWorkersLock.Lock()
for _, worker := range rpcWorkers {
    _ = worker.Close()
}
rpcWorkersLock.Unlock()

fmt.Println("Cleanup done.")
}

}
```

**User**

in this code, is there a chance I am closing a listener too early or something because i am missing lines package main

```

import ( "bufio" "bytes" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net"
"net/rpc" "os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP | sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.FileLogFileChan chan string }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
// 1. Start the RPC Server ONCE (outside the loop) // We do not want to re-register "WorkerIps" every time
// causing "multiple registration" panic go func() { _ = rpc.Register(&workers) listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println("IntroducePort Error:", err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go processStdin(input)
// Connect to HyDFS once hydfsClient, err := rpc.Dial("tcp", "localhost:8011") if err != nil { fmt.Println("Failed to
connect to HyDFS client:", err) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers available") }
// Reset State ctx, cancel := context.WithCancel(context.Background()) var wg sync.WaitGroup
r.LogFileChan = make(chan string, 500) r.StartTime = time.Now()
// --- 1. SETUP LISTENERS --- tupleListener, err := net.Listen("tcp", TuplePort) if err != nil {
fmt.Println("TuplePort bind error:", err) cancel() continue }
globalRmListener, err := net.Listen("tcp", GlobalRMPort) if err != nil { fmt.Println("GlobalRM bind error:", err) _ =
tupleListener.Close() cancel() continue }
// --- 2. START LOGGER --- wg.Add(1) go func() { defer wg.Done() path := filepath.Join(homeDir,
"RainStormLogs", "RainStorm_" +r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ =
r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Started RainStorm Application\n")
writer := bufio.NewWriter(r.LogFile) defer func() { writer.Flush() r.LogFile.Close() }()
for { select { case <-ctx.Done(): // Drain remaining logs for len(r.LogFileChan) > 0 { s := <-r.LogFileChan
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
return case s, ok := <-r.LogFileChan: if !ok { return } writer.WriteString(time.Now().Format("2006-01-02
15:04:05") + ":" + s) writer.Flush() } }() // INITIATE NEW RAINSTORM APPLICATION //Global RM /* 1. open
listener for current task input rates from workers 2. check if autoscale is on, if it is -> 3. compare rates to see if
changes are needed 4. complete changes */ // --- 3. START GLOBAL RM --- appServer := rpc.NewServer() _ =
appServer.Register(&r)
wg.Add(1) go func() { defer wg.Done() // Sidecar to close listener on exit, breaking the Accept loop go func() {
<-ctx.Done(); globalRmListener.Close() }()
for { conn, err := globalRmListener.Accept() if err != nil { return // Listener closed } go
appServer.ServeConn(conn) }()
// --- 4. CONNECT TO WORKERS --- workers.I.RLock() rpcWorkers = make(map[string]*rpc.Client)
rpcWorkersLock.Lock() for _, workerIp := range workers.ips { worker, err := rpc.Dial("tcp",
workerIp.String()+AssignmentPort) if err != nil { continue } rpcWorkers[workerIp.String()] = worker }
workers.I.RUnlock() rpcWorkersLock.Unlock()

```

```

r.Lock = new(sync.RWMutex)
r.Lock.Lock()
r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages)
r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false
r.initWorker()
r.NextAvailableVM = 0
for i := range r.NumStages {
    r.TaskInformation[i] = make(map[int]*TaskInfo)
    for j := range r.NumTasksPerStage {
        r.addTask(i, j)
        r.NextTaskNum[i]++
    }
}
r.sendIpls()
r.Lock.Unlock()

// @TODO: read srcFile from HyDFS and send into system at Input Rate for this application // send stage -1 is
done once done reading from the file // read in from local; output on HyDFS var createReply
[]resources.AddFileReply err = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
    RemoteName: r.HydfsDestinationFileName,
    Content: make([]byte, 0),
}, &createReply)
if err != nil {
    fmt.Println(err.Error())
}

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))
if err != nil {
    fmt.Println("Unable to open src directory: " + err.Error())
}
// buffered write to HyDFS output file
outputChan := make(chan string, 500)
wg.Add(1)
go func() {
    defer wg.Done()
    buffer := bytes.Buffer{}
    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            reply = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content: buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
    for {
        select {
        case <-ctx.Done():
            // DRAIN: Write everything remaining in channel for len(outputChan) > 0
            for line := <-outputChan {
                flush()
            }
            return
        case line := <-outputChan:
            buffer.WriteString(line)
        }
    }
}
wg.Add(1)
go func() {
    defer wg.Done()
    defer inputFile.Close()
    scanner := bufio.NewScanner(inputFile)
    // ... [Your existing Snapshot Logic Here]
    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0)
    for tid, info := range r.TaskInformation[0] {
        firstStageTasks[tid] = info.Ip
        firstTaskList = append(firstTaskList, tid)
    }
    sort.Ints(firstTaskList)
    r.Lock.Unlock()
    tupleClients := make(map[string]*WorkerClient, numWorkers)
    var numProcessed float64 = 0
    startTime := time.Now()
    // Local start time for rate limiting
    readingChan := make(chan struct {
        line string
        lineNumber int
    }, 100)
    producerRoutine := func() {
        lineNumber := 0
        for scanner.Scan() {
            readingChan <- struct {
                line string
                lineNumber int
            }{
                line: scanner.Text(),
                lineNumber: lineNumber,
            }
            lineNumber++
        }
    }
    readingChan <- struct {
        line string
        lineNumber int
    }{
        line: "",
        lineNumber: -1,
    }
    r.Lock.Lock()
    r.DoneReading = true
    r.Lock.Unlock()
}

// Consumer Routine
for {
    select {
    case <-ctx.Done():
        // Cleanup clients for _, c := range tupleClients {
        c.Conn.Close()
        // Done reading
    }
    case tuple := <-readingChan:
        if tuple.lineNumber == -1 {
            r.sendStageCompletion(-1)
            for _, c := range tupleClients {
                c.Conn.Close()
            }
            return
        }
        // Update Logic
        updatedMap := <-r.Stage1UpdatesChan
        firstStageTasks = updatedMap
        firstTaskList = make([]int, 0)
        for k := range firstStageTasks {
            firstTaskList = append(firstTaskList, k)
        }
        sort.Ints(firstTaskList)
        default:
        }
    }
    // Panic Fix
    if len(firstTaskList) == 0 {
        time.Sleep(50 * time.Millisecond)
        readingChan <- tuple
        continue
    }
    // Routing Logic
    nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)]
    nextTaskIp := firstStageTasks[nextTask]
    client, ok := tupleClients[nextTaskIp.String()]
    if !ok {
        conn, err := net.Dial("tcp", nextTaskIp.String() + TuplePort)
        if err != nil {
            // Retry logic
            delete(tupleClients, nextTaskIp.String())
            continue
        }
        client = &WorkerClient{
            Conn: conn,
            Buf: bufio.NewReader(conn),
        }
        tupleClients[nextTaskIp.String()] = client
    }
}

```

```

// ... [Sending Logic] ... _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) _, _ =
fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
// Wait ACK _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err :=
client.Buf.ReadString('\n') expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
if err != nil || strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients,
nextTaskIp.String()) readingChan <- tuple continue }
// Rate Limit numProcessed++ expectedDuration := time.Duration((numProcessed / r.InputRate) *
float64(time.Second)) targetTime := startTime.Add(expectedDuration) if targetTime.After(time.Now()) {
time.Sleep(targetTime.Sub(time.Now())) } } }() // needs to wait for the application to complete before cleaning
up --> @TODO: come back to this // --- 9. WAIT FOR COMPLETION AND CLEANUP --- <-appCompletedChan
fmt.Println("App completed Cleaning up")
// Stop all background routines cancel()
// Wait for Listeners, Writers, Logger to finish wg.Wait()
// Close workers rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
}

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.I.Lock() defer workers.I.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.I.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.I.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] =
ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d
op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply,
string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //@TODO: write to leader logs
when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n",
args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage
app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan
<- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages {
appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
}

```

```

rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } } }
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip:
workers.ips[app.NextAvailableVM%numWorkers]} workers.l.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } task := Task{
TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil {
fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)) }
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }
deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) } }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil {
fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
```

```
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true } break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break case "kill_task": //@TODO: add implementation for this break case "list_tasks": break }
```

## Gemini

Yes, you are absolutely closing the listener/writer too early relative to each other. This is a classic **"Producer-Consumer Shutdown Race."**

## The Problem

You are losing lines because of two specific race conditions in your shutdown sequence:

1. **The "Select" Roulette:** Inside your Tuple Listener, you have this:

```
Go
```

```
select {
    case outputChan <- line:
    case <-ctx.Done():
        return // ----- DATA LOSS HERE
}
```

When `ctx.Done()` is closed, if you have a `line` ready to send, Go's `select` picks a case at random. If it picks `ctx.Done()`, **you throw away the line you just read** and exit.

2. **The Writer Early Exit:** Your HyDFS writer listens for `ctx.Done()`. When that triggers, it drains whatever is currently in the channel buffer and exits.

- **The Race:** Your Tuple Listener is still running (cleaning up connections). It might successfully manage to send one last line to `outputChan`. But if the Writer has already processed `ctx.Done()` and exited, that line sits in the channel buffer forever and is never written to disk.

## The Fix: Ordered Shutdown

You cannot shut everything down at the exact same time (`cancel()`). You must shut down in stages:

1. **Stop Producers** (Cancel Context -> Tuple Listener stops).
2. **Wait for Producers** to finish sending.
3. **Close Channel** (Signals to Writer that no more data is coming).
4. **Wait for Writer** to finish draining.

Here is the corrected code with the **Ordered Shutdown Pipeline**.

```
Go
```



```
package main

import (
    "bufio"
    "bytes"
    "context"
    "fmt"
    ". \"g14-mp4/RainStorm/resources\""
    "g14-mp4/mp3/resources"
    "net"
    "net/rpc"
    "os"
    "path/filepath"
    "sort"
    "strconv"
    "strings"
    "sync"
    "time"
)

// ... [Structs WorkerIps, RainStorm, etc. remain unchanged] ...

func main() {
    homeDir, _ := os.UserHomeDir()
    dataDir = filepath.Join(homeDir, "data")
    workers = WorkerIps{}
    appCompletedChan = make(chan bool, 1)

    // RPC Register (Run once)
    go func() {
        _ = rpc.Register(&workers)
        listener, err := net.Listen("tcp", IntroducePort)
        if err != nil {
            fmt.Println("IntroducePort Error:", err)
            return
        }
        rpc.Accept(listener)
    }()

    input := make(chan RainStorm)
    go processStdin(input)
    hydfsClient, err := rpc.Dial("tcp", "localhost:8011")
    if err != nil {
        fmt.Println("Failed to connect to HyDFS:", err)
    }
    defer hydfsClient.Close()
```

```
for {
    r := <-input
    if numWorkers == 0 {
        panic("No workers")
    }

    // Context for cancellation (Stops Listeners/Input)
    ctx, cancel := context.WithCancel(context.Background())

    // WaitGroup for PRODUCERS only (Listeners, Input)
    var wgProducers sync.WaitGroup

    r.LogFileChan = make(chan string, 100)
    r.StartTime = time.Now()

    tupleListener, err := net.Listen("tcp", TuplePort)
    if err != nil {
        continue
    }

    // --- LOGGER ---
    // (Logger has its own simple shutdown, this is fine)
    go func() {
        path := filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("2006-01-02 15:04:05"))
        _, _ = os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755)
        r.LogFile, _ = os.OpenFile(path, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666)
        _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02 15:04:05") + ": Start")

        writer := bufio.NewWriter(r.LogFile)
        defer func() {
            writer.Flush()
            r.LogFile.Close()
        }()
    }

    for {
        select {
        case <-ctx.Done():
            // Drain channel then exit
            for len(r.LogFileChan) > 0 {
                s := <-r.LogFileChan
                writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
            }
            writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm")
            return
        case s, ok := <-r.LogFileChan:
```

```
        if !ok { return }
        writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s)
        writer.Flush()
    }
}

// --- GLOBAL RM ---
appServer := rpc.NewServer()
_ = appServer.Register(&r)
globalRmListener, err := net.Listen("tcp", GlobalRMPort)
if err != nil {
    continue
}

go func() {
    // Force close listener on context cancel
    go func() { <-ctx.Done(); globalRmListener.Close() }()
    for {
        conn, err := globalRmListener.Accept()
        if err != nil { return }
        go appServer.ServeConn(conn)
    }
}()

// ... [Dialing Workers logic remains unchanged] ...
workers.l.RLock()
rpcWorkers = make(map[string]*rpc.Client)
rpcWorkersLock.Lock()
for _, workerIp := range workers.ips {
    worker, err := rpc.Dial("tcp", workerIp.String()+AssignmentPort)
    if err != nil {
        continue
    }
    rpcWorkers[workerIp.String()] = worker
}
workers.l.RUnlock()
rpcWorkersLock.Unlock()

// ... [Initialize Structs logic remains unchanged] ...
r.Lock = new(sync.RWMutex)
r.Lock.Lock()
r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages)
r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false
```

```
r.initWorker()
r.NextAvailableVM = 0
for i := range r.NumStages {
    r.TaskInformation[i] = make(map[int]*TaskInfo)
    for j := range r.NumTasksPerStage {
        r.addTask(i, j)
        r.NextTaskNum[i]++
    }
}
r.sendIps()
r.Lock.Unlock()

// ... [HyDFS File Create logic remains unchanged] ...
var createReply []resources.AddFileReply
_ = hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
    RemoteName: r.HydfsDestinationFileName,
    Content:    make([]byte, 0),
}, &createReply)

inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))
if err != nil {
    fmt.Println("Unable to open src directory: " + err.Error())
}

// --- HYDFS BUFFERED WRITER (CONSUMER) ---
// Use a separate channel to signal when the WRITER is totally done
writerDone := make(chan bool)
outputChan := make(chan string, 500)

go func() {
    buffer := bytes.Buffer{}
    flush := func() {
        if buffer.Len() > 0 {
            var reply []resources.AppendReply
            _ = hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{
                RemoteName: r.HydfsDestinationFileName,
                Content:    buffer.Bytes(),
            }, &reply)
            buffer.Reset()
        }
    }
}

// Range loop runs until channel is CLOSED and EMPTY.
// It does NOT stop on ctx.Done(). This prevents data loss.
for line := range outputChan {
    buffer.WriteString(line)
```

```
        if buffer.Len() > 4096 {
            flush()
        }
    }
    // Flush remainder
    flush()
    // Signal main thread that we are finished
    close(writerDone)
}()

// --- TUPLE LISTENER (PRODUCER 1) ---
wgProducers.Add(1)
go func() {
    defer wgProducers.Done()

    go func() {
        <-ctx.Done()
        tupleListener.Close()
    }()
}

for {
    conn, err := tupleListener.Accept()
    if err != nil {
        return
    }

    wgProducers.Add(1) // Track individual connections
    go func(c net.Conn) {
        defer wgProducers.Done()
        defer c.Close()

        // Sidecar closer
        go func() { <-ctx.Done(); c.Close() }()

        reader := bufio.NewReader(c)
        for {
            line, err := reader.ReadString('\n')

            // FIX 1: Always prioritize sending data if we got it
            if len(line) > 0 {
                fmt.Println(line)
                outputChan <- line // Send. We know writer is alive until we close c
            }

            if err != nil {
                return
            }
        }
    }()
}
```

```
        }
    }
}(conn)
}

// --- INPUT READER (PRODUCER 2) ---
wgProducers.Add(1)
go func() {
    defer wgProducers.Done()
    defer inputFile.Close()

    scanner := bufio.NewScanner(inputFile)

    // ... [Snapshot logic] ...
    r.Lock.Lock()
    firstStageTasks := make(map[int]net.IP)
    firstTaskList := make([]int, 0)
    for tid, info := range r.TaskInformation[0] {
        firstStageTasks[tid] = info.Ip
        firstTaskList = append(firstTaskList, tid)
    }
    sort.Ints(firstTaskList)
    r.Lock.Unlock()

    tupleClients := make(map[string]*WorkerClient, numWorkers)
    var numProcessed float64 = 0
    startTime := time.Now()

    readingChan := make(chan struct{ line string; lineNumber int }, 100)

    go func() {
        lineNumber := 0
        for scanner.Scan() {
            readingChan <- struct{ line string; lineNumber int }{line: scanner.Text(), lineNumber: lineNumber}
            lineNumber++
        }
        readingChan <- struct{ line string; lineNumber int }{line: "", lineNumber: -1}
        r.Lock.Lock()
        r.DoneReading = true
        r.Lock.Unlock()
    }()
}

for {
    select {
    case <-ctx.Done():

```

```
        for _, c := range tupleClients { c.Conn.Close() }
        return

    case tuple := <-readingChan:
        if tuple.lineNum == -1 {
            r.sendStageCompletion(-1)
            for _, c := range tupleClients { c.Conn.Close() }
            return
        }

        // ... [Update Logic] ...
        select {
    case updatedMap := <-r.Stage1UpdatesChan:
        firstStageTasks = updatedMap
        firstTaskList = make([]int, 0)
        for k := range firstStageTasks {
            firstTaskList = append(firstTaskList, k)
        }
        sort.Ints(firstTaskList)
    default:
        }

        // Fix Divide By Zero
        if len(firstTaskList) == 0 {
            time.Sleep(50 * time.Millisecond)
            readingChan <- tuple
            continue
        }

        nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
        nextTaskIp := firstStageTasks[nextTask]

        client, ok := tupleClients[nextTaskIp.String()]
        if !ok {
            conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
            if err != nil {
                delete(tupleClients, nextTaskIp.String())
                continue
            }
            client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
            tupleClients[nextTaskIp.String()] = client
        }

        _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
        _, _ = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.lineNum,
```

```
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

if err != nil || strings.TrimSpace(ack) != expectedAck {
    client.Conn.Close()
    delete(tupleClients, nextTaskIp.String())
    readingChan <- tuple
    continue
}

numProcessed++
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
targetTime := startTime.Add(expectedDuration)
if targetTime.After(time.Now()) {
    time.Sleep(targetTime.Sub(time.Now())))
}
}
}
}
}()

// --- WAIT FOR APP COMPLETION ---
<-appCompletedChan
fmt.Println("App completed. Shutting down...")

// 1. Stop Producers (Listeners and Input Reader)
cancel()

// 2. Wait for Producers to completely finish
// This ensures the Listener has read every last byte from the workers
// and pushed it into outputChan.
wgProducers.Wait()

// 3. Now that no one is writing, we can safely close the channel.
close(outputChan)

// 4. Wait for the Writer to finish flushing the remaining buffer.
<-writerDone

// 5. Cleanup Resources
rpcWorkersLock.Lock()
for _, worker := range rpcWorkers {
    _ = worker.Close()
}
rpcWorkersLock.Unlock()
```

```
    fmt.Println("Cleanup done.")  
}  
}
```

**User**

I am going to attach my leader below. For some reason upon failure, the worker is getting the call that there is a failure and the restart occurs in mthe leaders logs. However, for some reason the system gets stuck and all rates go to 0. Is this a leader issue?

```
package main
import ("bufio" "bytes" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net"
"net/rpc" "os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP I sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.Time LogFile *os.FileLogFile Chan chan string }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string var curApp
*RainStorm
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
// RPC Register (Run once) go func() { _ = rpc.Register(&workers) listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println("IntroducePort Error:", err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") if
err != nil { fmt.Println("Failed to connect to HyDFS:", err) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } curApp = &r // Context for cancellation (Stops
Listeners/Input) ctx, cancel := context.WithCancel(context.Background())
// WaitGroup for PRODUCERS only (Listeners, Input) var wgProducers sync.WaitGroup
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now()
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { continue }
// --- LOGGER --- // (Logger has its own simple shutdown, this is fine) go func() { path :=
filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755) r.LogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02
15:04:05") + ": Started RainStorm Application\n")
writer := bufio.NewWriter(r.LogFile) defer func() { writer.Flush() r.LogFile.Close() }()
for { select { case <-ctx.Done(): // Drain channel then exit for len(r.LogFileChan) > 0 { s := <-r.LogFileChan
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s) }
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
return case s, ok := <-r.LogFileChan: if !ok { return } writer.WriteString(time.Now().Format("2006-01-02
15:04:05") + ": " + s) writer.Flush() } } }()
// --- GLOBAL RM --- appServer := rpc.NewServer() _ = appServer.Register(&r) globalRmListener, err :=
net.Listen("tcp", GlobalRMPort) if err != nil { continue }
go func() { // Force close listener on context cancel go func() { <-ctx.Done(); globalRmListener.Close() }() for {
conn, err := globalRmListener.Accept() if err != nil { return } go appServer.ServeConn(conn) } }()
// ... [Dialing Workers logic remains unchanged] ... workers.I.RLock() rpcWorkers =
make(map[string]*rpc.Client) rpcWorkersLock.Lock() for _, workerIp := range workers.ips { worker, err :=
rpc.Dial("tcp", workerIp.String()+AssignmentPort) if err != nil { continue } rpcWorkers[workerIp.String()] = worker } workers.I.RUnlock() rpcWorkersLock.Unlock()
r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] =
make(map[int]*TaskInfo) for j := range r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps()
```

```

r.Lock.Unlock()
// ... [HyDFS File Create logic remains unchanged] ... var createReply []resources.AddFileReply _ =
hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply)
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) }
// --- HYDFS BUFFERED WRITER (CONSUMER) --- // Use a separate channel to signal when the WRITER is
totally done writerDone := make(chan bool) outputChan := make(chan string, 500)
go func() { buffer := bytes.Buffer{} flush := func() { if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } }
// Range loop runs until channel is CLOSED and EMPTY. // It does NOT stop on ctx.Done(). This prevents data
loss. for line := range outputChan { buffer.WriteString(line) if buffer.Len() > 4096 { flush() } } // Flush remainder
flush() // Signal main thread that we are finished close(writerDone) }()
// --- TUPLE LISTENER (PRODUCER 1) --- wgProducers.Add(1) go func() { defer wgProducers.Done()
go func() { <-ctx.Done() tupleListener.Close() }()
for { conn, err := tupleListener.Accept() if err != nil { return }
wgProducers.Add(1) // Track individual connections go func(c net.Conn) { defer wgProducers.Done() defer
c.Close()
// Sidecar closer go func() { <-ctx.Done(); c.Close() }()
reader := bufio.NewReader(c) for { line, err := reader.ReadString('\n')
// FIX 1: Always prioritize sending data if we got it if len(line) > 0 { fmt.Print(line) outputChan <- line // Send. We
know writer is alive until we close chan. }
if err != nil { return } }(conn) }()
// --- INPUT READER (PRODUCER 2) --- wgProducers.Add(1) go func() { defer wgProducers.Done() defer
inputFile.Close()
scanner := bufio.NewScanner(inputFile)
r.Lock.Lock() firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range
r.TaskInformation[0] { firstStageTasks[tid] = info.ip firstTaskList = append(firstTaskList, tid) }
sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) var numProcessed float64 = 0 startTime :=
time.Now()
readingChan := make(chan struct { line string lineNumber int }, 100)
go func() { lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line:
scanner.Text(), lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true r.Lock.Unlock() }()
for { select { case <-ctx.Done(): for _, c := range tupleClients { c.Conn.Close() } return
case tuple := <-readingChan: if tuple.lineNumber == -1 { r.sendStageCompletion(-1) for _, c := range tupleClients
{ c.Conn.Close() } return }
// ... [Update Logic] ... select { case updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap
firstTaskList = make([]int, 0) for k := range firstStageTasks { firstTaskList = append(firstTaskList, k) }
sort.Ints(firstTaskList) default: }
// Fix Divide By Zero if len(firstTaskList) == 0 { time.Sleep(50 * time.Millisecond) readingChan <- tuple
continue }
nextTask := firstTaskList[tuple.lineNumber%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { delete(tupleClients, nextTaskIp.String()) continue } client = &WorkerClient{Conn: conn, Buf:
bufio.NewReader(conn)} tupleClients[nextTaskIp.String()] = client }
_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) _, _ = fmt.Fprintf(client.Conn, "%s-
%d,%d,%d,%s\n", "temp", tuple.lineNumber, 0, nextTask, tuple.line)
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNumber, "ACK")

```

```

if err != nil || strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients,
nextTaskIp.String()) readingChan <- tuple continue }
numProcessed++ expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second))
targetTime := startTime.Add(expectedDuration) if targetTime.After(time.Now()) {
time.Sleep(targetTime.Sub(time.Now())) } } } }()
// --- WAIT FOR APP COMPLETION --- <-appCompletedChan fmt.Println("App completed. Cleaning up")
// 1. Stop Producers (Listeners and Input Reader) cancel()
// 2. Wait for Producers to completely finish // This ensures the Listener has read every last byte from the
workers // and pushed it into outputChan. wgProducers.Wait()
// 3. Now that no one is writing, we can safely close the channel. close(outputChan)
// 4. Wait for the Writer to finish flushing the remaining buffer. <-writerDone
// 5. Cleanup Resources rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
fmt.Println("Cleanup done.") } }
func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.I.Lock() defer workers.I.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.I.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.I.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d
op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(), reply,
string(app.Ops[task.Stage].Name)) app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //TODO: write to leader logs
when receiving a tuple rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n",
args.Rate, args.Task, args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage
app.Lock.Lock() taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan
<- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] == 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages {
appCompletedChan <- true } }
} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }
func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }
func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIPs",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <

```

```

numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " + x.Error.Error()) } }
func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current app is modified
    waitingChan := make(chan *rpc.Call, len(rpcWorkers))
    numSuccess := 0
    rpcWorkersLock.RLock()
    args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile: app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers {
        var reply int
        worker.Go("Worker.Initialize", args, &reply, waitingChan)
        numSuccess++
    }
    rpcWorkersLock.RUnlock()
    for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }
}
func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
    //if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
    workers.l.RLock()
    app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%numWorkers]}
    workers.l.RUnlock()
    //app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
    app.NextAvailableVM++
    if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp }
    task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
    var reply int
    rpcWorkersLock.RLock()
    rpcWorker := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()]
    rpcWorkersLock.RUnlock()
    err := rpcWorker.Call("Worker.AddTask", task, &reply)
    if err != nil { fmt.Println("Failed to send request to add task: " + err.Error()) }
    app.TaskInformation[stageNum][taskNum].Pid = reply //TODO: also log the local logfile on the task
    app.LogFileChan <- fmt.Sprintf("Starting Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply, string(app.Ops[stageNum].Name))
}
func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED
    if len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return }
    var taskNum int
    for k := range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized because of GO
        taskNum = k
        break
    }
    deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum]
    if !exists { fmt.Printf("Failed to remove task: %d, stage %d: not exists", taskNum, stageNum)
        return
    }
    delete(app.TaskInformation[stageNum], taskNum)
    if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp }
    app.sendIps()
    task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
    var reply int
    rpcWorkersLock.RLock()
    rpcWorker := rpcWorkers[deletedTaskIp.String()]
    rpcWorkersLock.RUnlock()
    err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply)
    if err != nil { fmt.Println("Failed to send request to kill task: " + err.Error()) }
}
func processStdin(i1 chan<- RainStorm) {
    scanner := bufio.NewScanner(os.Stdin)
    for {
        scanner.Scan()
        stdIn := scanner.Text()
        stdIn = strings.TrimSpace(stdIn)
        splits := strings.Split(stdIn, " ")
        switch splits[0] {
        case "RainStorm": // var rainStorm RainStorm
            bad := false
            for i := 1; i < len(splits) && !bad; i++ {
                var err error
                switch {
                case i == 1: // NumStages
                    rainStorm.NumStages, err = strconv.Atoi(splits[i])
                    if err != nil { fmt.Println("Failed to parse NumStages: " + err.Error()) }
                    bad = true
                case i == 2: // NumTasksPerStage
                    rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i])
                    if err != nil { fmt.Println("Failed to parse NumTasksPerStage: " + err.Error()) }
                    bad = true
                case i == len(splits)-7: // processing operations and arguments
                    rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args: splits[i+1]})
                }
            }
        }
        break
    }
}

```

```

strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this vm := splits[1] pid, _ := strconv.Atoi(splits[2])
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { done := false for taskNum, info :=
range stage { if vm == info.Ip.String() && pid == info.Pid { rpcWorkersLock.RLock() worker :=
rpcWorkers[info.Ip.String()] var reply int _ = worker.Call("Worker.KillTask", TaskID{ Task: taskNum, Stage:
stageNum, }, &reply) rpcWorkersLock.RUnlock() done = true break } } if done { break } }
curApp.Lock.RUnlock() break
case "list_tasks": //@TODO print local log file for task curApp.Lock.RLock() for stageNum, stage := range
curApp.TaskInformation { for _, info := range stage { fmt.Printf("%s %d %s\n", info.Ip.String(), info.Pid,
curApp.Ops[stageNum]) } } curApp.Lock.RUnlock() break
} } }

```

## Gemini

Yes, this is absolutely a **Leader Deadlock** issue.

## The Diagnosis

You are experiencing a "Circular Dependency Deadlock" between your Control Plane (RPCs) and your Data Plane (Input Loop).

- The Trigger:** A worker fails. The Input Loop tries to send a tuple, but the network call hangs (waiting for timeout/error). **The Input Loop stops reading from channels while it waits.**
- The Lock:** The Leader detects the failure and calls `ReceiveFailure`. This function **acquires the global `app.Lock`**.
- The Block:** Inside `ReceiveFailure`, you try to send the new topology to `app.Stage1UpdatesChan`.
- The Deadlock:** Because the Input Loop is stuck waiting on the failed worker, it isn't reading from the channel. The channel buffer fills up. **ReceiveFailure blocks forever** trying to push to the channel, while holding `app.Lock`.
- The Symptom:** Since `ReceiveFailure` is holding the lock, **ReceiveRateUpdate cannot acquire the lock**. All incoming rate updates pile up and eventually time out or are ignored, causing your reported rates to drop to 0.

## The Fix

You must use a **Non-Blocking Select** when sending updates to the input loop. If the input loop is busy, we skip sending the update (this is safe, because the next update will carry the correct state anyway).

Here are the **3 functions** you need to replace in your code to fix this.

### 1. Fix `ReceiveFailure`

Go



```

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()
}

```

```

if _, exists := app.TaskInformation[task.Stage][task.TaskNumber]; !exists {
    fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
} else {
    workers.l.RLock()
    // Reassign IP
    app.TaskInformation[task.Stage][task.TaskNumber].Ip = workers.ips[app.NextAvailableVM]
    workers.l.RUnlock()

    app.NextAvailableVM++

    if task.Stage == 0 && !app.DoneReading {
        temp := make(map[int]net.IP)
        for t, ip := range app.TaskInformation[0] {
            temp[t] = ip.Ip
        }

        // --- FIX: NON-BLOCKING SEND ---
        // Prevent Leader from freezing if Input Loop is busy
        select {
        case app.Stage1UpdatesChan <- temp:
        default:
            // Channel full/Input busy. Skip update to release lock.
        }
        // -----
    }
}

app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d op_exe: %s\n", app
// Note: This calls Worker.AddTask. If that worker is slow, this might still hold the
app.addTask(task.Stage, task.TaskNumber)
app.sendIps()
}

return nil
}

```

## 2. Fix addTask

Go



```

func (app *RainStorm) addTask(stageNum int, taskNum int) {
    // Note: Caller holds Lock
    workers.l.RLock()
    app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM]}
    workers.l.RUnlock()
}

```

```

app.NextAvailableVM++


if stageNum == 0 && !app.DoneReading {
    temp := make(map[int]net.IP)
    for task, ip := range app.TaskInformation[0] {
        temp[task] = ip.Ip
    }

    // --- FIX: NON-BLOCKING SEND ---
    select {
    case app.Stage1UpdatesChan <- temp:
    default:
    }
    // -----
}

task := Task{
    TaskNumber: taskNum,
    Stage:      stageNum,
    Executable: app.Ops[stageNum],
}

var reply int
rpcWorkersLock.RLock()
// Check if worker exists before dialing to prevent panic
if _, ok := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()]; ok {
    rpcWorker := rpcWorkers[app.TaskInformation[stageNum][taskNum].Ip.String()]
    err := rpcWorker.Call("Worker.AddTask", task, &reply)
    if err != nil {
        fmt.Println("Failed to send request to add task: " + err.Error())
    }
    app.TaskInformation[stageNum][taskNum].Pid = reply
    app.LogFileChan <- fmt.Sprintf("Starting Task at VM: %s PID: %d op_exe: %s\n", app.
} else {
    fmt.Println("Error: Worker IP not found in rpcWorkers map")
}
rpcWorkersLock.RUnlock()
}

```

### 3. Fix `removeTask`

Go



```

func (app *RainStorm) removeTask(stageNum int) {
    // Note: Caller holds Lock
}

```

```
if len(app.TaskInformation[stageNum]) <= 1 {
    return
}

var taskNum int
for k := range app.TaskInformation[stageNum] {
    taskNum = k
    break
}

deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum]
if !exists {
    return
}

delete(app.TaskInformation[stageNum], taskNum)

if stageNum == 0 && !app.DoneReading {
    temp := make(map[int]net.IP)
    for task, ip := range app.TaskInformation[0] {
        temp[task] = ip.Ip
    }

    // --- FIX: NON-BLOCKING SEND ---
    select {
    case app.Stage1UpdatesChan <- temp:
    default:
    }
    // -----
}
}

app.sendIps()

task := Task{
    TaskNumber: taskNum,
    Stage:      stageNum,
    Executable: app.Ops[stageNum],
}

var reply int
rpcWorkersLock.RLock()
if rpcWorker, ok := rpcWorkers[deletedTaskIp.Ip.String()]; ok {
    err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply)
    if err != nil {
        fmt.Println("Failed to send request to kill task: " + err.Error())
    }
}
```

```
    rpcWorkersLock.RUnlock()  
}
```

## User

but the deadlock never occurred but the send stage completion for stage -1 went out

## Gemini

The issue is specifically caused by the `!app.DoneReading` check in your `ReceiveFailure` function. This flag is being set **too early**, causing your Leader to stop updating its routing table while it is still trying to send data.

## The Sequence of Events Causing the "Stuck" State

1. **Scanner Finishes Fast:** Your `go func()` scanning the file runs very fast (memory speed). It finishes and sets `r.DoneReading = true`.
2. **Sender is Still Working:** Your input loop (Consumer) is still sending tuples over the network. This process is much slower and might take minutes to finish.
3. **Failure Happens:** A Stage 0 worker fails while the Sender is still working.
4. **Leader Blocks Update:** `ReceiveFailure` runs. It checks `if task.Stage == 0 && !app.DoneReading`.
  - Since `DoneReading` is **true** (from step 1), the Leader **skips sending the new IP map to Stage1UpdatesChan**.
5. **Sender Gets Stuck:** The Input Sender doesn't get the update. It keeps trying to send tuples to the **Old Dead IP**. It retries indefinitely, blocking the entire stream.
6. **Result:** All rates drop to 0 because the source is stuck trying to talk to a ghost.

## The Fix

You must remove the `!app.DoneReading` check so the Input Sender always gets the latest topology.

**However**, to do this safely without causing a deadlock (Leader Freeze), you **must** use a non-blocking `select` statement.

### 1. Update `ReceiveFailure`

Remove `&& !app.DoneReading` and use `select/default`.

Go



```
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    if _, exists := app.TaskInformation[task.Stage][task.TaskNumber]; !exists {
        fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
    } else {
        workers.l.RLock()
        app.TaskInformation[task.Stage][task.TaskNumber].Ip = workers.ips[app.NextAvailable]
        workers.l.RUnlock()

        app.NextAvailableVM++

        // FIX: Removed "!app.DoneReading" check.
    }
}
```

```
// If the input loop is running, it needs this update.  
// If it is NOT running, the select/default ensures we don't hang.  
if task.Stage == 0 {  
    temp := make(map[int]net.IP)  
    for t, ip := range app.TaskInformation[0] {  
        temp[t] = ip.Ip  
    }  
  
    // NON-BLOCKING SEND  
    select {  
        case app.Stage1UpdatesChan <- temp:  
        default:  
            // Channel full or no reader. Safe to skip.  
    }  
}  
  
app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d\n", app.TaskInform  
app.addTask(task.Stage, task.TaskNumber)  
app.sendIps()  
}  
return nil  
}
```

## 2. Update addTask

Apply the same logic: Remove the flag check, use non-blocking send.

```
func (app *RainStorm) addTask(stageNum int, taskNum int) {  
    // ... existing setup ...  
  
    // FIX: Removed !app.DoneReading  
    if stageNum == 0 {  
        temp := make(map[int]net.IP)  
        for task, ip := range app.TaskInformation[0] {  
            temp[task] = ip.Ip  
        }  
        // FIX: Non-blocking send  
        select {  
            case app.Stage1UpdatesChan <- temp:  
            default:  
        }  
    }  
}
```

```
// ... existing RPC call ...  
}
```

### 3. Update `removeTask`

Apply the same logic here as well.

Go



```
func (app *RainStorm) removeTask(stageNum int) {  
    // ... existing delete logic ...  
  
    delete(app.TaskInformation[stageNum], taskNum)  
  
    // FIX: Removed !app.DoneReading and used select  
    if stageNum == 0 {  
        temp := make(map[int]net.IP)  
        for task, ip := range app.TaskInformation[0] {  
            temp[task] = ip.Ip  
        }  
        select {  
            case app.Stage1UpdatesChan <- temp:  
            default:  
        }  
        app.sendIps()  
  
        // ... existing RPC call ...  
    }  
}
```

## Summary of what this solves

- 1. Stuck Stream:** By removing the flag, your input loop now learns about the new worker IP immediately and resumes sending.
- 2. No Deadlocks:** By using `select { default: }`, you ensure that if the input loop *has* actually finished and stopped reading the channel, the Leader won't freeze trying to write to it.

**User**

ok but the task i failed was a stage 1 task

**Gemini**

This is a great observation. If "Stage completion for stage -1" went out, it means your code set `r.DoneReading = true` inside the scanner goroutine.

**Here is the exact bug causing your "Stuck" state:**

1. **Scanner Finishes Fast:** Your `scanner` reads the file quickly and sets `r.DoneReading = true`.
2. **Network is Slow:** Your Input Loop (Consumer) is still churning through those lines, sending them one by one. This might take minutes.
3. **Failure Occurs:** A Stage 0 worker fails while the Consumer is still working.
4. **Leader Logic Error:** `ReceiveFailure` runs. It checks:

Go



```
if task.Stage == 0 && !app.DoneReading { ... }
```

Since `DoneReading` is **true** (from Step 1), the Leader **skips sending the new IP** to `Stage1UpdatesChan`.

5. **Result:** The Input Loop never receives the new IP. It keeps retrying the **Dead IP** forever. The system hangs, and rates drop to 0.

**The Fix**

You must **remove the `!app.DoneReading` check** from `ReceiveFailure`, `addTask`, and `removeTask`. The Input Loop *always* needs to know about topology changes as long as it is running.

To do this safely (without deadlocking if the loop *has* actually finished), you simply use the **Non-Blocking Select** pattern we discussed.

**Apply this fix to `ReceiveFailure`, `addTask`, and `removeTask`:**

**1. Fixed `ReceiveFailure`**

Go



```
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    if _, exists := app.TaskInformation[task.Stage][task.TaskNumber]; !exists {
        fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
    } else {
        workers.l.RLock()
        app.TaskInformation[task.Stage][task.TaskNumber].Ip = workers.ips[app.NextAvailable]
        workers.l.RUnlock()

        app.NextAvailableVM++
    }
}
```

```

// FIX: Removed "!app.DoneReading".
// We MUST send the update if it's Stage 0, regardless of reading status.
if task.Stage == 0 {
    temp := make(map[int]net.IP)
    for t, ip := range app.TaskInformation[0] {
        temp[t] = ip.Ip
    }
    // NON-BLOCKING SEND avoids deadlock if the reader is actually gone
    select {
    case app.Stage1UpdatesChan <- temp:
    default:
    }
}
app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d\n", app.TaskInformation[0].VmName, app.TaskInformation[0].Pid)
app.addTask(task.Stage, task.TaskNumber)
app.sendIps()
}
return nil
}

```

## 2. Fixed `addTask`

Go



```

func (app *RainStorm) addTask(stageNum int, taskNum int) {
    // ... [Locking logic] ...

    // FIX: Removed !app.DoneReading
    if stageNum == 0 {
        temp := make(map[int]net.IP)
        for task, ip := range app.TaskInformation[0] {
            temp[task] = ip.Ip
        }
        select {
        case app.Stage1UpdatesChan <- temp:
        default:
        }
    }

    // ... [RPC logic] ...
}

```

## 3. Fixed `removeTask`

Go



```
func (app *RainStorm) removeTask(stageNum int) {
    // ... [Selection logic] ...

    delete(app.TaskInformation[stageNum], taskNum)

    // FIX: Removed !app.DoneReading
    if stageNum == 0 {
        temp := make(map[int]net.IP)
        for task, ip := range app.TaskInformation[0] {
            temp[task] = ip.Ip
        }
        select {
        case app.Stage1UpdatesChan <- temp:
        default:
        }
    }
    // ... [Rest of function] ...
}
```

**User**

here is my updated leader. when killing a task i am sending the correct PID's information to the worker. Is there a reason the PID is different from the one I am killing when i am printing when restarting when the leader receives the failure

```

package main
import ("bufio" "bytes" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net"
"het/rpc" "os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP I sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.Time LogFile *os.File LogFileChan chan string
FinalTuplesLock sync.RWMutex FinalTuples map[string]bool }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string var curApp
*RainStorm
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
// RPC Register (Run once) go func() { _ = rpc.Register(&workers) listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println("IntroducePort Error:", err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") if
err != nil { fmt.Println("Failed to connect to HyDFS:", err) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } curApp = &r // Context for cancellation (Stops
Listeners/Input) ctx, cancel := context.WithCancel(context.Background())
// WaitGroup for PRODUCERS only (Listeners, Input) var wgProducers sync.WaitGroup
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now() r.FinalTuples = make(map[string]bool)
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { continue }
// --- LOGGER --- // (Logger has its own simple shutdown, this is fine) go func() { path :=
filepath.Join(homeDir, "RainStormLogs", "RainStorm_" + r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755) rLogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = rLogFile.WriteString(r.StartTime.Format("2006-01-02
15:04:05") + ": Started RainStorm Application\n")
writer := bufio.NewWriter(rLogFile) defer func() { writer.Flush() rLogFile.Close() }()
for { select { case <-ctx.Done(): // Drain channel then exit for len(r.LogFileChan) > 0 { s := <-r.LogFileChan
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ":" + s) }
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
return case s, ok := <-r.LogFileChan: if !ok { return } writer.WriteString(time.Now().Format("2006-01-02
15:04:05") + ":" + s) writer.Flush() } } }()
// --- GLOBAL RM --- appServer := rpc.NewServer() _ = appServer.Register(&r) globalRmListener, err :=
net.Listen("tcp", GlobalRMPort) if err != nil { continue }
go func() { // Force close listener on context cancel go func() { <-ctx.Done(); globalRmListener.Close() } }()
for { conn, err := globalRmListener.Accept() if err != nil { return } go appServer.ServeConn(conn) } }()
// ... [Dialing Workers logic remains unchanged] ... workers.l.RLock() rpcWorkers =
make(map[string]*rpc.Client) rpcWorkersLock.Lock() for _, workerIp := range workers.ips { worker, err :=
rpc.Dial("tcp", workerIp.String()+AssignmentPort) if err != nil { continue } rpcWorkers[workerIp.String()] = worker
} workers.l.RUnlock() rpcWorkersLock.Unlock()
r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 20)
r.DoneReading = false r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] =
```

```

make(map[int]*TaskInfo) for j := range r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps()
r.Lock.Unlock()
// ... [HyDFS File Create logic remains unchanged] ... var createReply []resources.AddFileReply _ =
hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) os.MkdirAll(filepath.Join(homeDir,
"RainStormOutputs"), 0755) localOutputFile, _ := os.OpenFile(filepath.Join(homeDir, "RainStormOutputs",
r.HydfsDestinationFileName), os.O_CREATE|os.O_TRUNC|os.O_WRONLY, 0666)
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) }
// --- HYDFS BUFFERED WRITER (CONSUMER) --- // Use a separate channel to signal when the WRITER is
totally done writerDone := make(chan bool) outputChan := make(chan string, 500)
go func() { buffer := bytes.Buffer{} flush := func() { if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } }
// Range loop runs until channel is CLOSED and EMPTY. // It does NOT stop on ctx.Done(). This prevents data
loss. for line := range outputChan { buffer.WriteString(line) if buffer.Len() > 4096 { flush() } } // Flush remainder
flush() // Signal main thread that we are finished close(writerDone) }()
// --- TUPLE LISTENER (PRODUCER 1) --- wgProducers.Add(1) go func() { defer wgProducers.Done()
go func() { <-ctx.Done() tupleListener.Close() }()
for { conn, err := tupleListener.Accept() if err != nil { return }
wgProducers.Add(1) go func(c net.Conn) { defer wgProducers.Done() defer c.Close()
// Sidecar closer go func() { <-ctx.Done(); c.Close() }()
reader := bufio.NewReader(c) for { // 1. Read the message line, err := reader.ReadString('\n') if err != nil { return
}
line = strings.TrimSpace(line) if len(line) == 0 { continue }
// 2. Parse Protocol: "TaskID-TupleID,Content" // We split by the first comma only parts := strings.SplitN(line,
",", 2) if len(parts) < 2 { fmt.Println("Invalid tuple format received:", line) continue }
id := parts[0] // e.g., "1:4-50" content := parts[1] // e.g., "Count: 100"
// 3. Deduplication Check r.FinalTuplesLock.Lock() seen := r.FinalTuples[id] if !seen { r.FinalTuples[id] = true // //
Mark as seen } r.FinalTuplesLock.Unlock()
// 4. Write to File (Only if NEW) if !seen { localOutputFile.WriteString(content + "\n") // Add newline back
outputChan <- content + "\n" }
// 5. Send ACK (ALWAYS send ACK, even if duplicate) // The worker is waiting for this to stop retrying. //
Format: "TaskID-TupleID-ACK" ackMsg := fmt.Sprintf("%s-ACK\n", id) _, _ = c.Write([]byte(ackMsg)) } }(conn)
} }()
// --- INPUT READER (PRODUCER 2) --- wgProducers.Add(1) go func() { defer wgProducers.Done() defer
inputFile.Close()
scanner := bufio.NewScanner(inputFile)
r.Lock.Lock() firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range
r.TaskInformation[0] { firstStageTasks[tid] = info.ip firstTaskList = append(firstTaskList, tid) }
sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) var numProcessed float64 = 0 startTime :=
time.Now()
readingChan := make(chan struct { line string lineNumber int }, 100)
go func() { lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line:
scanner.Text(), lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} r.Lock.Lock() r.DoneReading = true r.Lock.Unlock() }()
for { select { case <-ctx.Done(): for _, c := range tupleClients { c.Conn.Close() } return
case tuple := <-readingChan: if tuple.lineNumber == -1 { r.sendStageCompletion(-1) for _, c := range tupleClients
{ c.Conn.Close() } return }
// ... [Update Logic] ... select { case updatedMap := <-r.Stage1UpdatesChan: firstStageTasks = updatedMap
firstTaskList = make([]int, 0) for k := range firstStageTasks { firstTaskList = append(firstTaskList, k) }
}

```

```

sort.Ints(firstTaskList) default: }
// Fix Divide By Zero if len(firstTaskList) == 0 { time.Sleep(50 * time.Millisecond) readingChan <- tuple
continue }

nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)] nextTaskIp := firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { delete(tupleClients, nextTaskIp.String()) continue } client = &WorkerClient{Conn: conn, Buf:
bufio.NewReader(conn)} tupleClients[nextTaskIp.String()] = client }

_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) _, _ = fmt.Fprintf(client.Conn, "%s-
%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, err := client.Buf.ReadString("\n")
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
if err != nil || strings.TrimSpace(ack) != expectedAck { client.Conn.Close() delete(tupleClients,
nextTaskIp.String()) readingChan <- tuple continue }

numProcessed++ expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second))
targetTime := startTime.Add(expectedDuration) if targetTime.After(time.Now()) {
time.Sleep(targetTime.Sub(time.Now())) } } }()

// --- WAIT FOR APP COMPLETION --- <-appCompletedChan
// 1. Stop Producers (Listeners and Input Reader) cancel()
// 2. Wait for Producers to completely finish // This ensures the Listener has read every last byte from the
workers // and pushed it into outputChan. wgProducers.Wait()
// 3. Now that no one is writing, we can safely close the channel. close(outputChan)
// 4. Wait for the Writer to finish flushing the remaining buffer. <-writerDone
// 5. Cleanup Resources rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
localOutputFile.Close() fmt.Println("RainStorm Application completed") } }

func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }

func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] = ip.Ip }
app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task because of Failure
at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(),
app.TaskInformation[task.Stage][task.TaskNumber].Pid, string(app.Ops[task.Stage].Name))
app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil }

func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error { //@TODO: write to leader logs when receiving a tuple
rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n", args.Rate, args.Task,
args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock()
taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <-
fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists :=
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID:
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { //
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { }
```

```

appCompletedChan <- true } }

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT
should not have received this\n", args.Task) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIps",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } }

func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.Lock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip:
workers.ips[app.NextAvailableVM%numWorkers]} workers.l.Unlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } task := Task{
TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err != nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name))

func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k := range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }

deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }
delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } app.sendIps()

task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill
task: " + err.Error()) }

func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case

```

```
"RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch {
case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to
parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage
rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse
NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations
and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args:
splits[i+1]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i]
break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i]
break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err !=
nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: //
AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse
.AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err =
strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true }
break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err
!= nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: //
HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to
parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this vm := splits[1] pid, _ := strconv.Atoi(splits[2])
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { done := false for taskNum, info := range
stage { if vm == info.Ip.String() && pid == info.Pid { rpcWorkersLock.RLock() worker := rpcWorkers[info.Ip.String()]
var reply int _ = worker.Call("Worker.KillTask", TaskID{ Task: taskNum, Stage: stageNum, }, &reply)
rpcWorkersLock.RUnlock() done = true break } } if done { break } }
curApp.Lock.RUnlock() break
case "list_tasks": //@TODO print local log file for task curApp.Lock.RLock() for stageNum, stage := range
curApp.TaskInformation { for _, info := range stage { fmt.Printf("%s %d %s\n", info.Ip.String(), info.Pid,
curApp.Ops[stageNum]) } } curApp.Lock.RUnlock() break
} } }
```

## Gemini

Yes, there is a very specific and correct reason for this.

**A PID (Process ID) is tied to a specific instance of a running process.** When you kill a task, that process dies. You cannot "revive" a dead process ID.

To restart the task, your worker must spawn a **brand new process** (or goroutine). The Operating System assigns a **new PID** to this new process.

Here is the flow of events in your code:

1. **Kill:** You kill PID `100`. Process `100` ceases to exist.
2. **ReceiveFailure:** The Leader updates the struct to point to a new Worker IP.
3. **Log:** You log "Restarting Task...". At this exact moment, your struct still holds the **OLD PID** (`100`).
4. **AddTask:** You call the worker. The worker starts a **NEW** process. The OS assigns it PID `101`.
5. **Reply:** The worker returns `101` to the leader.
6. **Update:** The leader updates the struct to `101`.

So, seeing a different PID upon restart is **proof that your system is working correctly**.

## ⚠ CRITICAL WARNING: You Re-introduced the "Stuck" Bug

In the code you just pasted, you have **re-added the bug** that causes the system to hang and rates to drop to 0 when a Stage 0 task fails.

In `ReceiveFailure`, `addTask`, and `removeTask`, you have this line:

**Go**

```
if task.Stage == 0 && !app.DoneReading { // <--- THIS BUG IS BACK
```

### Why this breaks your system:

1. Your file scanner finishes quickly and sets `DoneReading = true`.
2. Your Sender loop is still working (sending tuples).
3. A Stage 0 Worker fails.
4. `ReceiveFailure` sees `DoneReading` is true, so it **SKIPS** sending the new IP to the Sender loop.
5. The Sender loop keeps sending tuples to the **Dead IP** forever.

### The Fix (Apply Immediately)

You must remove `&& !app.DoneReading` and use `select` to safely send the update.

#### 1. Fix `ReceiveFailure`

**Go**

```
// FIX: Removed !app.DoneReading check
if task.Stage == 0 {
    temp := make(map[int]net.IP)
    for t, ip := range app.TaskInformation[0] {
        temp[t] = ip.Ip
    }
    // FIX: Use Non-Blocking Select
    select {
        case app.Stage1UpdatesChan <- temp:
        default:
    }
}
```

#### 2. Fix `addTask`

**Go**

```
// FIX: Removed !app.DoneReading check
if stageNum == 0 {
    temp := make(map[int]net.IP)
    for task, ip := range app.TaskInformation[0] {
        temp[task] = ip.Ip
    }
    select {
        case app.Stage1UpdatesChan <- temp:
        default:
```

```
    }  
}
```

### 3. Fix `removeTask`

Go



```
// FIX: Removed !app.DoneReading check  
if stageNum == 0 {  
    temp := make(map[int]net.IP)  
    for task, ip := range app.TaskInformation[0] {  
        temp[task] = ip.Ip  
    }  
    select {  
        case app.Stage1UpdatesChan <- temp:  
        default:  
    }  
}
```

**User**

but before spinning it back up i am logging that I am restarting the task based on the old PID

**Gemini**

Yes, that is exactly correct logic.

You are logging "Restarting Task..." **before** you call `app.addTask()`.

At that specific line of code in `ReceiveFailure`, your `app.TaskInformation` struct still contains the **Old PID** (the one that just failed). You haven't generated the new one yet because the RPC to the worker hasn't happened.

**The Timeline of your Logs**

1. **Event:** Task fails (Old PID: 100).

2. `ReceiveFailure` runs:

- You choose a new IP.
- **Log:** "Restarting Task... PID: 100" (You are logging the *state at time of failure*).

3. `app.addTask` runs:

- Calls Worker RPC.
- Worker starts new process (New PID: 101).
- Worker returns `101`.
- **Update:** You update `app.TaskInformation` to `101`.
- **Log:** "Starting Task... PID: 101".

This creates a perfect audit trail: `Restarting PID 100` -> `Starting PID 101`.

**FINAL CRITICAL FIX**

I must warn you one last time: **You re-introduced the bug that causes your system to hang.**

In the code you just pasted, you have this line in `ReceiveFailure`, `addTask`, and `removeTask`:

Go



```
if task.Stage == 0 && !app.DoneReading { // <--- THIS WILL FREEZE YOUR SYSTEM
```

If a Stage 0 worker fails **after** the file is done reading (but while the sender is still working), the Leader will **not** send the new IP to the sender. The sender will keep trying to talk to the dead worker forever, and your rate will drop to 0.

**You MUST change it to this in all 3 functions:**

Go



```
// FIX: Remove "!app.DoneReading"
if task.Stage == 0 {
    temp := make(map[int]net.IP)
    for t, ip := range app.TaskInformation[0] {
```

```
    temp[t] = ip.Ip
}
// FIX: Use select/default so it never deadlocks
select {
case app.Stage1UpdatesChan <- temp:
default:
}
}
```

**User**

why is there a default case

**Gemini**

The `default` case is what makes the channel operation **Non-Blocking**.

Without the `default` case, if the channel is full, your program stops right there and waits. With the `default` case, if the channel is full, your program says "never mind" and keeps moving.

Here is why this is critical for your Leader:

## 1. The Scenario: "The Full Buffer"

Imagine your `Stage1UpdatesChan` has a buffer size of 20.

- Your Input Loop is temporarily stuck (e.g., waiting 3 seconds for a network timeout from a failed worker).
- While it is stuck, it stops reading from the channel.
- Meanwhile, you have a burst of activity (failures, scaling) that calls `addTask` 21 times.
- The channel fills up.

## 2. Without Default (The Crash)

Go



```
select {
case app.Stage1UpdatesChan <- temp:
    // If channel is full, CODE BLOCKS HERE FOREVER.
}
```

- `addTask` goes to sleep, waiting for a slot to open.
- **Crucial Detail:** `addTask` is currently holding `app.Lock`.
- Because the lock is held, **no other RPCs can run**. `ReceiveRateUpdate`, `ReceiveTaskCompletion`, and `ReceiveFailure` all freeze waiting for the lock.
- **Result:** Your Leader is deadlocked.

## 3. With Default (The Fix)

Go



```
select {
case app.Stage1UpdatesChan <- temp:
    // Sent successfully
default:
    // Channel was full. We skip the send and CONTINUE.
    // We release the lock and let the Leader keep working.
}
```

- If the channel is full, Go immediately executes the `default` block.

- `addTask` finishes, releases `app.Lock`, and the Leader stays alive.

## Is it safe to skip updates?

**Yes.** Because you are sending the **entire map** (a snapshot state), not a "delta" (a change event).

- **Update 1:** {Task 0: IP\_A} (Skipped because full)
- **Update 2:** {Task 0: IP\_B} (Sent successfully)

When the Input Loop finally wakes up and reads **Update 2**, it has the correct current state ( IP\_B ). It doesn't matter that it missed the intermediate state ( IP\_A ).

## Next Step

Would you like me to generate a `run.sh` script or a `Makefile` to help you automate the building and running of your Leader and Workers for testing?

**User**

keep the blocking , I do not want anything to be dropped

**Gemini**

You are asking to keep the channel send **blocking** because you are worried about losing topology updates.

I will provide the **blocking code below**, but I must clarify one critical concept: "**Dropping**" a map update is **not data loss**. Since you are sending the *entire* map every time (a snapshot), the Input Loop only ever needs the *latest* map. If the channel is full, it means the Input Loop already has a pending update queue. Skipping an intermediate update is actually a performance optimization called **State Coalescing**.

However, per your request, here is the implementation using **Blocking Sends**.

## The Fix for "System Getting Stuck"

The reason your system gets stuck is **not** because updates are dropped, but because of the `!app.DoneReading` check.

1. Your file scanner finishes quickly -> sets `DoneReading = true`.
2. Your sender is still working (slowly).
3. A failure happens.
4. Leader sees `DoneReading == true` and **skips** sending the new IP.
5. Sender loops forever trying to talk to the dead IP.

You must remove `!app.DoneReading` to fix the stuck system.

## Corrected Leader Logic (Blocking Sends + Fix)

Apply this pattern to `ReceiveFailure`, `addTask`, and `removeTask`.

### 1. Fixed `ReceiveFailure`

```
Go 
```

```
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    if _, exists := app.TaskInformation[task.Stage][task.TaskNumber]; !exists {
        fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber, task.Stage)
    } else {
        // ... (IP reassignment logic) ...

        workers.l.RLock()
        app.TaskInformation[task.Stage][task.TaskNumber].Ip = workers.ips[app.NextAvailableVm]
        workers.l.RUnlock()
        app.NextAvailableVm++

        // FIX 1: Removed "!app.DoneReading".
        // The Input Loop needs this update even if the *Scanner* is finished.
```

```
if task.Stage == 0 {
    temp := make(map[int]net.IP)
    for t, ip := range app.TaskInformation[0] {
        temp[t] = ip.Ip
    }

    // FIX 2: Blocking Send (As requested)
    // CAUTION: If the input loop has exited (finished job), this line will
    // BLOCK FOREVER and freeze your Leader.
    // Ensure your channel buffer (20) is large enough to handle shutdown races.
    app.Stage1UpdatesChan <- temp
}

app.LogFileChan <- fmt.Sprintf("Restarting Task at VM: %s PID: %d\n", app.TaskInform
app.addTask(task.Stage, task.TaskNumber)
app.sendIps()
}
return nil
}
```

## 2. Fixed `addTask`

Go 

```
func (app *RainStorm) addTask(stageNum int, taskNum int) {
    // ... (locking and setup) ...

    // FIX 1: Removed "!app.DoneReading"
    if stageNum == 0 {
        temp := make(map[int]net.IP)
        for task, ip := range app.TaskInformation[0] {
            temp[task] = ip.Ip
        }
        // FIX 2: Blocking Send
        app.Stage1UpdatesChan <- temp
    }

    // ... (RPC calls) ...
}
```

## 3. Fixed `removeTask`

Go 

```
func (app *RainStorm) removeTask(stageNum int) {
    // ... (delete logic) ...

    // FIX 1: Removed "!app.DoneReading"
    if stageNum == 0 {
        temp := make(map[int]net.IP)
        for task, ip := range app.TaskInformation[0] {
            temp[task] = ip.Ip
        }
        // FIX 2: Blocking Send
        app.Stage1UpdatesChan <- temp
    }
    app.sendIps()

    // ... (RPC calls) ...
}
```

## Recommendation to Prevent Deadlocks with Blocking

Since you are using blocking sends, if your Input Loop finishes processing the file and exits, there is **no one reading the channel**. If `addTask` tries to write to it, the Leader will freeze.

To support blocking sends safely, you should **increase the channel buffer** in `main` to be larger than the maximum number of updates you expect during a shutdown sequence.

Go



```
// In main()
// Increase buffer from 20 to 100 or more to prevent freezing on completion
r.Stage1UpdatesChan = make(chan map[int]net.IP, 100)
```

**User**

but then how will the sending go routine know it is finished

**Gemini**

You are asking exactly the right question. This is the crux of the logic problem.

The reason your system got "stuck" before was not because of blocking vs. non-blocking. It was because you were setting the "I am finished" flag **too early**.

## The Timing Problem

1. **Scanner Routine:** Reads the file into memory. This is fast. It finishes and sets `DoneReading = true`.
2. **Sender Routine:** Takes those lines and sends them over the network. This is slow. It is still working **long after** the scanner finished.
3. **The Bug:** Because `DoneReading` is true, the Leader stops sending updates to the Sender. The Sender is still running, blindly trying to talk to a failed worker, and gets stuck.

## The Solution: Move the Flag

You must move the `r.DoneReading = true` line. Do not set it when the **Scanner** finishes. Set it when the **Sender** finishes.

This ensures that as long as the Sender loop is alive and processing tuples, `DoneReading` remains `false`, and the Leader will correctly (and blockingly) send updates.

## Corrected Code Blocks

Here is how to modify your `processStdin` / `main` loop logic.

### 1. Remove the flag from the Scanner (Producer)

Go to your scanner goroutine inside `main` and **delete** the lock/bool setting lines.

Go



```
// Producer Routine
go func() {
    lineNumber := 0
    for scanner.Scan() {
        readingChan <- struct {
            line    string
            lineNumber int
        }{line: scanner.Text(), lineNumber: lineNumber}
        lineNumber++
    }
    readingChan <- struct {
        line    string
        lineNumber int
    }{line: "", lineNumber: -1}
}
```

```
// DELETE THESE LINES FROM HERE:  
// r.Lock.Lock()  
// r.DoneReading = true  
// r.Lock.Unlock()  
}()
```

## 2. Add the flag to the Sender (Consumer)

Move that logic to the very end of your Sender loop, right before it calls `r.sendStageCompletion(-1)`.

Go



```
// Consumer Routine  
for {  
    select {  
        case <-ctx.Done():  
            // ... cleanup ...  
            return  
  
        case tuple := <-readingChan:  
            if tuple.lineNum == -1 {  
                // --- ADD IT HERE ---  
                // The sender is officially done processing.  
                // Now it is safe to tell the Leader to stop sending updates.  
                r.Lock.Lock()  
                r.DoneReading = true  
                r.Lock.Unlock()  
                // -----  
  
                r.sendStageCompletion(-1)  
                for _, c := range tupleClients { c.Conn.Close() }  
                return  
            }  
  
            // ... rest of loop ...
```

## 3. Keep your Blocking Sends in Leader

Now that the flag accurately reflects the lifecycle of the Sender, you can safely use your blocking logic in `ReceiveFailure`, `addTask`, and `removeTask`.

Go



```
// In ReceiveFailure, addTask, etc.  
  
if task.Stage == 0 {
```

```
// We check the flag.  
// If False: The sender is still running, so we MUST send the update (Block is safe).  
// If True: The sender has exited (or is exiting), so we skip (Block avoided).  
if !app.DoneReading {  
    temp := make(map[int]net.IP)  
    for t, ip := range app.TaskInformation[0] {  
        temp[t] = ip.Ip  
    }  
    // BLOCKING SEND (User Request)  
    app.Stage1UpdatesChan <- temp  
}  
}
```

## Why this works

1. **While sending:** `DoneReading` is false. Leader sends updates. `Stage1UpdatesChan` is read by the active sender. **No Deadlock**.
2. **When finished:** Sender sets `DoneReading` to true and exits. Leader stops trying to write to the channel. **No Deadlock**.
3. **The Race Condition:** There is a tiny moment where Leader sees `false`, then Sender sets `true` and exits, then Leader writes.
  - **Safety Net:** Your channel buffer ( 20 ). The Leader writes to the buffer. The value sits there ignored (since Sender is dead). Leader finishes. **No Deadlock**.

This setup grants your wish: **No dropped updates while the system is active, and no deadlocks when the system finishes.**

**User**

wait, but what about sends that needed to be retried?

**Gemini**

This is a sharp catch. You have identified a **critical data loss bug** in the "Re-queue" logic.

## The "Re-queue" Bug

If you put a failed tuple back into `readingChan` (`readingChan <- tuple`), it goes to the **back of the line**.

1. **Scanner:** Pushes `Tuple 100`, then pushes `EOF (-1)`.
2. **Sender:** Reads `Tuple 100`.
3. **Network:** Fails.
4. **Sender:** Pushes `Tuple 100` back into `readingChan`.
5. **Channel State:** `[EOF, Tuple 100]`.
6. **Sender:** Reads next item... it gets `EOF`.
7. **Sender:** Sees `-1`, sets `DoneReading=true`, and exits.
8. **Result:** `Tuple 100` is left in the channel and **never sent**.

## The Solution: The "Retry Loop"

You must not put the tuple back in the channel. Instead, if a failure occurs, you must enter a tiny internal loop that keeps trying to send **that specific tuple** until it succeeds.

Crucially, inside this retry loop, you must **continue listening for topology updates** from the Leader.

## The Fixed Sender Routine

Replace your `// Consumer Routine` block with this logic. It solves both the data ordering issue and the deadlock issue.

```
Go   
  
// Consumer Routine  
for {  
    select {  
        case <-ctx.Done():  
            for _, c := range tupleClients { c.Conn.Close() }  
            return  
  
        case tuple := <-readingChan:  
            if tuple.lineNum == -1 {  
                // Only set flag and exit when we legitimately hit the end  
                r.Lock.Lock()  
                r.DoneReading = true  
                r.Lock.Unlock()  
  
                r.sendStageCompletion(-1)  
            }  
    }  
}
```

```
        for _, c := range tupleClients { c.Conn.Close() }
        return
    }

    // --- RETRY LOOP ---
    // We stay inside this loop until THIS tuple is successfully sent.
    for {
        // 1. Check for Topology Updates (Critical for retries!)
        select {
            case updatedMap := <-r.Stage1UpdatesChan:
                firstStageTasks = updatedMap
                firstTaskList = make([]int, 0)
                for k := range firstStageTasks {
                    firstTaskList = append(firstTaskList, k)
                }
                sort.Ints(firstTaskList)
            default:
        }

        // 2. Safety Check
        if len(firstTaskList) == 0 {
            time.Sleep(50 * time.Millisecond)
            continue // Retry loop
        }

        // 3. Routing
        nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
        nextTaskIp := firstStageTasks[nextTask]

        // 4. Connection
        client, ok := tupleClients[nextTaskIp.String()]
        if !ok {
            conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
            if err != nil {
                // Connection failed. Wait and retry loop.
                time.Sleep(100 * time.Millisecond)
                continue
            }
            client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
            tupleClients[nextTaskIp.String()] = client
        }

        // 5. Send & Ack
        _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
        _, err := fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple.li
        if err == nil {
```

```
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n')
expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")

if err == nil && strings.TrimSpace(ack) == expectedAck {
    // SUCCESS! Break the retry loop and process next tuple
    break
}
}

// 6. Failure Handling (Inside Retry Loop)
// If we are here, send or ack failed.
fmt.Println("Failed to send tuple, retrying...", err)
client.Conn.Close()
delete(tupleClients, nextTaskIp.String())

// Do NOT return or break.
// Loop back to top of 'for', check for updates, and try again.
}

// 7. Rate Limiting (Only after success)
numProcessed++
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
targetTime := startTime.Add(expectedDuration)
if targetTime.After(time.Now()) {
    time.Sleep(targetTime.Sub(time.Now())))
}
}
}
```

## Why this works

- Topology Updates:** The inner retry loop checks `Stage1UpdatesChan`. If the Leader sends a new IP because of the failure, this loop picks it up immediately.
- No Deadlock:** Because we haven't hit the `-1` tuple yet, `DoneReading` is false. The Leader sees this and happily sends the update (even with blocking sends).
- Correct Order:** We never push the tuple back into `readingChan`, so it never gets stuck behind the `EOF` marker. We process strictly 1, 2, 3... EOF.

## User

I am using this , however i keep getting the following error in an infinite loop : Failed to send tuple, retrying...  
<nil>

## Gemini

The reason you see <nil> is due to a **Variable Shadowing Bug** in Go.

Inside your `if` block, you use `ack, err := ...`. The `:=` operator declares a `new err` variable that only exists inside that curly brace scope. When the code falls through to the failure block, the `fmt.Println(..., err)` prints the `outer err` (from `Fprintf`), which was `nil`.

This is masking the real error, which is almost certainly a **Read Timeout** or a **Bad ACK**.

## The Fix

You need to decouple the error checking logic so you can see the *real* reason it failed.

Replace your **Consumer Routine** (specifically the inner retry loop) with this corrected version. I have fixed the shadowing bug and improved the logging so you can see exactly why it is failing.

Go



```
// Consumer Routine
for {
    select {
    case <-ctx.Done():
        for _, c := range tupleClients { c.Conn.Close() }
        return

    case tuple := <-readingChan:
        if tuple.lineNum == -1 {
            r.Lock.Lock()
            r.DoneReading = true
            r.Lock.Unlock()

            r.sendStageCompletion(-1)
            for _, c := range tupleClients { c.Conn.Close() }
            return
        }

    // --- RETRY LOOP ---
    for {
        // 1. Check for Topology Updates
        select {
        case updatedMap := <-r.Stage1UpdatesChan:
            firstStageTasks = updatedMap
            firstTaskList = make([]int, 0)
            for k := range firstStageTasks {
                firstTaskList = append(firstTaskList, k)
            }
        }
    }
}
```

```
        }

        sort.Ints(firstTaskList)

    default:
        }

        if len(firstTaskList) == 0 {
            time.Sleep(50 * time.Millisecond)
            continue
        }

        // 2. Routing & Connection
        nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
        nextTaskIp := firstStageTasks[nextTask]

        client, ok := tupleClients[nextTaskIp.String()]
        if !ok {
            conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
            if err != nil {
                fmt.Println("Dial failed, retrying:", err) // Print real dial err
                time.Sleep(100 * time.Millisecond)
                continue
            }
            client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
            tupleClients[nextTaskIp.String()] = client
        }

        // 3. Send & Ack (Fixed Shadowing)
        var ack string
        var sendErr, ackErr error // Separate errors to avoid confusion

        _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
        _, sendErr = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n", "temp", tuple

        if sendErr == nil {
            _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
            ack, ackErr = client.Buf.ReadString('\n')
        }

        // 4. Validation
        expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
        isAckValid := (ackErr == nil) && (strings.TrimSpace(ack) == expectedAck

        if sendErr == nil && isAckValid {
            // SUCCESS! Break loop.
            break
        }
```

```
// 5. Failure Handling
// Now we print the SPECIFIC error that caused the fail
if sendErr != nil {
    fmt.Println("Send Error:", sendErr)
} else if ackErr != nil {
    fmt.Println("Ack Read Error:", ackErr) // Likely "i/o timeout" or "
} else {
    fmt.Printf("Invalid Ack. Expected: '%s', Got: '%s'\n", expectedAck,
}

client.Conn.Close()
delete(tupleClients, nextTaskIp.String())

// Loop back and try again (hopefully pulling a new IP from updates)
}

// Rate Limiting
numProcessed++
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
targetTime := startTime.Add(expectedDuration)
if targetTime.After(time.Now()) {
    time.Sleep(targetTime.Sub(time.Now())))
}
}

}
```

## What to expect now

Run this code. Instead of `<nil>`, you will likely see one of these:

- **"Ack Read Error: i/o timeout"**: The worker is receiving the tuple but not sending an ACK back within 3 seconds. Check your worker logic.
- **"Ack Read Error: EOF"**: The worker is closing the connection immediately after receiving the tuple.
- **"Invalid Ack..."**: The worker is sending data back, but it's not the string `"temp-NUM-ACK"`.

**User**

nice fix so here is my new updated code. I am working on the worker as well but could it be from the leader (pasted below) that could be causing the program to get stuck after receiving a failure

package main

```
import ( "bufio" "bytes" "context" "fmt" . "g14-mp4/RainStorm/resources" "g14-mp4/mp3/resources" "net"
"net/rpc" "os" "path/filepath" "sort" "strconv" "strings" "sync" "time" )
type WorkerIps struct { ips []net.IP I sync.RWMutex }
type RainStorm struct { NumStages int NumTasksPerStage int HydfsSrcDirectory string
HydfsDestinationFileName string ExactlyOnce bool AutoScale bool InputRate float64 LowestRate float64
HighestRate float64 Ops []Operation TaskInformation []map[int]*TaskInfo // [stage][task] --> IP, PID
NextTaskNum []int // [stage] NextAvailableVM int Stage1UpdatesChan chan map[int]net.IP Lock
*sync.RWMutex DoneReading bool StartTime time.TimeLogFile *os.FileLogFileChan chan string
FinalTuplesLock sync.RWMutex FinalTuples map[string]bool }
const clientTimeout = time.Second * 3
var workers WorkerIps var numWorkers int var numSuccessfulDials int var rpcWorkers map[string]*rpc.Client
var rpcWorkersLock sync.RWMutex var appCompletedChan chan bool var dataDir string var curApp
*RainStorm
func main() { homeDir, _ := os.UserHomeDir() dataDir = filepath.Join(homeDir, "data") workers = WorkerIps{}
appCompletedChan = make(chan bool, 1)
// RPC Register (Run once) go func() { _ = rpc.Register(&workers) listener, err := net.Listen("tcp",
IntroducePort) if err != nil { fmt.Println("IntroducePort Error:", err) return } rpc.Accept(listener) }()
input := make(chan RainStorm) go processStdin(input) hydfsClient, err := rpc.Dial("tcp", "localhost:8011") if
err != nil { fmt.Println("Failed to connect to HyDFS:", err) } defer hydfsClient.Close()
for { r := <-input if numWorkers == 0 { panic("No workers") } curApp = &r // Context for cancellation (Stops
Listeners/Input) ctx, cancel := context.WithCancel(context.Background())
// WaitGroup for PRODUCERS only (Listeners, Input) var wgProducers sync.WaitGroup
r.LogFileChan = make(chan string, 100) r.StartTime = time.Now() r.FinalTuples = make(map[string]bool)
tupleListener, err := net.Listen("tcp", TuplePort) if err != nil { continue }
// --- LOGGER --- // (Logger has its own simple shutdown, this is fine) go func() { path :=
filepath.Join(homeDir, "RainStormLogs", "RainStorm_"+r.StartTime.Format("20060102150405")) _ =
os.MkdirAll(filepath.Join(homeDir, "RainStormLogs"), 0755) r.LogFile, _ = os.OpenFile(path,
os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0666) _, _ = r.LogFile.WriteString(r.StartTime.Format("2006-01-02
15:04:05") + ": Started RainStorm Application\n")
writer := bufio.NewWriter(r.LogFile) defer func() { writer.Flush() r.LogFile.Close() }()
for { select { case <-ctx.Done(): // Drain channel then exit for len(r.LogFileChan) > 0 { s := <-r.LogFileChan
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": " + s) }
writer.WriteString(time.Now().Format("2006-01-02 15:04:05") + ": RainStorm Application Completed\n")
return case s, ok := <-r.LogFileChan: if !ok { return } writer.WriteString(time.Now().Format("2006-01-02
15:04:05") + ": " + s) writer.Flush() } } }()
// --- GLOBAL RM --- appServer := rpc.NewServer() _ = appServer.Register(&r) globalRmListener, err :=
net.Listen("tcp", GlobalRMPort) if err != nil { continue }
go func() { // Force close listener on context cancel go func() { <-ctx.Done(); globalRmListener.Close() }() for {
conn, err := globalRmListener.Accept() if err != nil { return } go appServer.ServeConn(conn) } }()
// ... [Dialing Workers logic remains unchanged] ... workers.I.RLock() rpcWorkers =
make(map[string]*rpc.Client) rpcWorkersLock.Lock() for _, workerIp := range workers.ips { worker, err :=
rpc.Dial("tcp", workerIp.String()+AssignmentPort) if err != nil { continue } rpcWorkers[workerIp.String()] = worker
} workers.I.RUnlock() rpcWorkersLock.Unlock()
r.Lock = new(sync.RWMutex) r.Lock.Lock() r.TaskInformation = make([]map[int]*TaskInfo, r.NumStages)
r.NextTaskNum = make([]int, r.NumStages) r.Stage1UpdatesChan = make(chan map[int]net.IP, 100)
r.DoneReading = false r.initWorker() r.NextAvailableVM = 0 for i := range r.NumStages { r.TaskInformation[i] =
make(map[int]*TaskInfo) for j := range r.NumTasksPerStage { r.addTask(i, j) r.NextTaskNum[i]++ } } r.sendIps()
```

```

r.Lock.Unlock()
// ... [HyDFS File Create logic remains unchanged] ... var createReply []resources.AddFileReply _ =
hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: make([]byte, 0), }, &createReply) os.MkdirAll(filepath.Join(homeDir,
"RainStormOutputs"), 0755) localOutputFile, _ := os.OpenFile(filepath.Join(homeDir, "RainStormOutputs",
r.HydfsDestinationFileName), os.O_CREATE|os.O_TRUNC|os.O_WRONLY, 0666)
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory)) if err != nil { fmt.Println("Unable to open
src directory: " + err.Error()) }
// --- HYDFS BUFFERED WRITER (CONSUMER) --- // Use a separate channel to signal when the WRITER is
totally done writerDone := make(chan bool) outputChan := make(chan string, 500)
go func() { buffer := bytes.Buffer{} flush := func() { if buffer.Len() > 0 { var reply []resources.AppendReply _ =
hydfsClient.Call("Client.RemoteAppend", &resources.RemoteFileArgs{ RemoteName:
r.HydfsDestinationFileName, Content: buffer.Bytes(), }, &reply) buffer.Reset() } }
// Range loop runs until channel is CLOSED and EMPTY. // It does NOT stop on ctx.Done(). This prevents data
loss. for line := range outputChan { buffer.WriteString(line) if buffer.Len() > 4096 { flush() } } // Flush remainder
flush() // Signal main thread that we are finished close(writerDone) }()
// --- TUPLE LISTENER (PRODUCER 1) --- wgProducers.Add(1) go func() { defer wgProducers.Done()
go func() { <-ctx.Done() tupleListener.Close() }()
for { conn, err := tupleListener.Accept() if err != nil { return }
wgProducers.Add(1) go func(c net.Conn) { defer wgProducers.Done() defer c.Close()
// Sidecar closer go func() { <-ctx.Done(); c.Close() }()
reader := bufio.NewReader(c) for { // 1. Read the message line, err := reader.ReadString('\n') if err != nil { return
}
line = strings.TrimSpace(line) if len(line) == 0 { continue }
// 2. Parse Protocol: "TaskID-TupleID,Content" // We split by the first comma only parts := strings.SplitN(line,
",", 2) if len(parts) < 2 { fmt.Println("Invalid tuple format received:", line) continue }
id := parts[0] // e.g., "1:4-50" content := parts[1] // e.g., "Count: 100"
// 3. Deduplication Check r.FinalTuplesLock.Lock() seen := r.FinalTuples[id] if !seen { r.FinalTuples[id] = true //
Mark as seen } r.FinalTuplesLock.Unlock()
// 4. Write to File (Only if NEW) if !seen { localOutputFile.WriteString(content + "\n") // Add newline back
outputChan <- content + "\n" }
// 5. Send ACK (ALWAYS send ACK, even if duplicate) // The worker is waiting for this to stop retrying. //
Format: "TaskID-TupleID-ACK" ackMsg := fmt.Sprintf("%s-ACK\n", id) _, _ = c.Write([]byte(ackMsg)) } }(conn)
} }()
// --- INPUT READER (PRODUCER 2) --- wgProducers.Add(1) go func() { defer wgProducers.Done() defer
inputFile.Close()
scanner := bufio.NewScanner(inputFile)
r.Lock.Lock() firstStageTasks := make(map[int]net.IP) firstTaskList := make([]int, 0) for tid, info := range
r.TaskInformation[0] { firstStageTasks[tid] = info.Ip firstTaskList = append(firstTaskList, tid) }
sort.Ints(firstTaskList) r.Lock.Unlock()
tupleClients := make(map[string]*WorkerClient, numWorkers) var numProcessed float64 = 0 startTime :=
time.Now()
readingChan := make(chan struct { line string lineNumber int }, 100)
go func() { lineNumber := 0 for scanner.Scan() { readingChan <- struct { line string lineNumber int }{line:
scanner.Text(), lineNumber: lineNumber} lineNumber++ } readingChan <- struct { line string lineNumber int }{line: "", lineNumber: -1} }()
// Consumer Routine // Consumer Routine for { select { case <-ctx.Done(): for _, c := range tupleClients {
c.Conn.Close() } return
case tuple := <-readingChan: if tuple.lineNum == -1 { r.Lock.Lock() r.DoneReading = true r.Lock.Unlock()
r.sendStageCompletion(-1) for _, c := range tupleClients { c.Conn.Close() } return }
// --- RETRY LOOP --- for { // 1. Check for Topology Updates select { case updatedMap := <-
r.Stage1UpdatesChan: firstStageTasks = updatedMap firstTaskList = make([]int, 0) for k := range

```

```

firstStageTasks { firstTaskList = append(firstTaskList, k) } sort.Ints(firstTaskList) default: }
if len(firstTaskList) == 0 { time.Sleep(50 * time.Millisecond) continue }
// 2. Routing & Connection nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)] nextTaskIp :=
firstStageTasks[nextTask]
client, ok := tupleClients[nextTaskIp.String()] if !ok { conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
if err != nil { fmt.Println("Dial failed, retrying:", err) // Print real dial error time.Sleep(100 * time.Millisecond)
continue } client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)} tupleClients[nextTaskIp.String()] =
client }
// 3. Send & Ack (Fixed Shadowing) var ack string var sendErr, ackErr error // Separate errors to avoid
confusion
_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout)) _, sendErr = fmt.Fprintf(client.Conn, "%s-
%d,%d,%d,%s\n", "temp", tuple.lineNum, 0, nextTask, tuple.line)
if sendErr == nil { _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout)) ack, ackErr =
client.Buf.ReadString('\n') }
// 4. Validation expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK") isAckValid := (ackErr ==
nil) && (strings.TrimSpace(ack) == expectedAck)
if sendErr == nil && isAckValid { // SUCCESS! Break loop. break }
// 5. Failure Handling // Now we print the SPECIFIC error that caused the fail if sendErr != nil { fmt.Println("Send
Error:", sendErr) } else if ackErr != nil { fmt.Println("Ack Read Error:", ackErr) // Likely "i/o timeout" or "EOF" }
else { fmt.Printf("Invalid Ack. Expected: '%s', Got: '%s'\n", expectedAck, strings.TrimSpace(ack)) }
client.Conn.Close() delete(tupleClients, nextTaskIp.String())
// Loop back and try again (hopefully pulling a new IP from updates) }
// Rate Limiting numProcessed++ expectedDuration := time.Duration((numProcessed / r.InputRate) *
float64(time.Second)) targetTime := startTime.Add(expectedDuration) if targetTime.After(time.Now()) {
time.Sleep(targetTime.Sub(time.Now())) } } } }()
// --- WAIT FOR APP COMPLETION --- <-appCompletedChan
// 1. Stop Producers (Listeners and Input Reader) cancel()
// 2. Wait for Producers to completely finish // This ensures the Listener has read every last byte from the
workers // and pushed it into outputChan. wgProducers.Wait()
// 3. Now that no one is writing, we can safely close the channel. close(outputChan)
// 4. Wait for the Writer to finish flushing the remaining buffer. <-writerDone
// 5. Cleanup Resources rpcWorkersLock.Lock() for _, worker := range rpcWorkers { _ = worker.Close() }
rpcWorkersLock.Unlock()
localOutputFile.Close() fmt.Println("RainStorm Application completed") } }
func (w *WorkerIps) AddWorker(args net.IP, reply *int) error { workers.l.Lock() defer workers.l.Unlock()
workers.ips = append(workers.ips, args) numWorkers++ return nil }
func (app *RainStorm) ReceiveFailure(task Task, reply *int) error { // restart the task on the next worker in the
cycle app.Lock.Lock() defer app.Lock.Unlock() if _, exists := app.TaskInformation[task.Stage]
[task.TaskNumber]; !exists { fmt.Printf("Failing task:%d at stage: %d does not exist", task.TaskNumber,
task.Stage) } else { workers.l.RLock() app.TaskInformation[task.Stage][task.TaskNumber].Ip =
workers.ips[app.NextAvailableVM%numWorkers] workers.l.RUnlock() app.NextAvailableVM++ if task.Stage ==
0 && !app.DoneReading { temp := make(map[int]net.IP) for t, ip := range app.TaskInformation[0] { temp[t] =
ip.Ip } app.Stage1UpdatesChan <- temp } app.LogFileChan <- fmt.Sprintf("Restarting Task because of Failure
at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[task.Stage][task.TaskNumber].Ip.String(),
app.TaskInformation[task.Stage][task.TaskNumber].Pid, string(app.Ops[task.Stage].Name))
app.addTask(task.Stage, task.TaskNumber) app.sendIps() } return nil } func (app *RainStorm)
ReceiveRateUpdate(args RmUpdate, reply *int) error { //@TODO: write to leader logs when receiving a tuple
rate //app.LogFile app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n", args.Rate, args.Task,
args.Stage) if app.AutoScale { if args.Rate < app.LowestRate { // add a task to this stage app.Lock.Lock()
taskNum := app.NextTaskNum[args.Stage] app.NextTaskNum[args.Stage]++ app.LogFileChan <-
fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate) app.addTask(args.Stage, taskNum)
app.sendIps() app.Lock.Unlock() } else if args.Rate > app.HighestRate { // remove a task from this stage
}
}

```

```

app.Lock.Lock() app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stage, args.Rate)
app.removeTask(args.Stage) app.Lock.Unlock() } } return nil }

func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error { //stage completion manager -->
manage markers from tasks saying they are done app.Lock.Lock() defer app.Lock.Unlock() if _, exists := 
app.TaskInformation[args.Stage][args.Task]; exists { app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: 
%d Stage: %d VM: %s PID: %d op_exe: %s\n", args.Task, args.Stage, app.TaskInformation[args.Stage]
[args.Task].Ip.String(), reply, string(app.Ops[args.Stage].Name)) delete(app.TaskInformation[args.Stage],
args.Task) //app.CurNumTasks[args.Stage] -= 1 app.sendIps() if len(app.TaskInformation[args.Stage]) == 0 { // 
stage completed app.sendStageCompletion(args.Stage) if args.Stage+1 == app.NumStages { 
appCompletedChan <- true } }

} else { //do nothing because this should never happen fmt.Printf("Received task completion for: %d, BUT 
should not have received this\n", args.Task) } return nil }

func (app *RainStorm) sendStageCompletion(completedStage int) { waitingChan := make(chan *rpc.Call,
len(rpcWorkers)) numSuccess := 0 rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int
worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan) numSuccess++ }
rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil {
fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error()) } } }

func (app *RainStorm) sendIps() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when current
app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() for _, worker := range rpcWorkers { var reply int worker.Go("Worker.ReceiveIps",
app.TaskInformation, &reply, waitingChan) numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i <
numSuccess; i++ { x := <-waitingChan if x.Error != nil { fmt.Println("Failed to send IPs to workers: " +
x.Error.Error()) } } }

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only called when
current app is modified waitingChan := make(chan *rpc.Call, len(rpcWorkers)) numSuccess := 0
rpcWorkersLock.RLock() args := InitArgs{ Ops: app.Ops, Time: app.StartTime, HyDFSDestFile:
app.HydfsDestinationFileName, LowWatermark: app.LowestRate, HighWatermark: app.HighestRate, } for _, 
worker := range rpcWorkers { var reply int worker.Go("Worker.Initialize", args, &reply, waitingChan)
numSuccess++ } rpcWorkersLock.RUnlock() for i := 0; i < numSuccess; i++ { x := <-waitingChan if x.Error != nil
{ fmt.Println("Failed to send list of operations to workers: " + x.Error.Error()) } } }

func (app *RainStorm) addTask(stageNum int, taskNum int) { //MUST BE WRAPPED IN LOCK WHEN CALLED
//if taskNum > app.StageCounter[stageNum]] { // app.TaskInformation[stageNum] =
append(app.TaskInformation[stageNum], workers.ips[app.NextAvailableVM%numWorkers]) //} else { //
app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers] //}
workers.l.RLock() app.TaskInformation[stageNum][taskNum] = &TaskInfo{ip:
workers.ips[app.NextAvailableVM%numWorkers]} workers.l.RUnlock()
//app.TaskCompletion[stageNum].StateTracker[taskNum] = false //app.NextTaskNum[stageNum]++
app.NextAvailableVM++ if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := 
range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } task := Task{
TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], }
var reply int rpcWorkersLock.RLock() rpcWorker := rpcWorkers[app.TaskInformation[stageNum]
[taskNum].Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AddTask", task, &reply) if err
!= nil { fmt.Println("Failed to send request to add task: " + err.Error()) } app.TaskInformation[stageNum]
[taskNum].Pid = reply //TODO: also log the local logfile on the task app.LogFileChan <- fmt.Sprintf("Starting
Task at VM: %s PID: %d op_exe: %s\n", app.TaskInformation[stageNum][taskNum].Ip.String(), reply,
string(app.Ops[stageNum].Name)) }

func (app *RainStorm) removeTask(stageNum int) { //MUST BE WRAPPED IN APP LOCK WHEN CALLED if
len(app.TaskInformation[stageNum]) <= 1 { // only 1 task remaining in the stage return } var taskNum int for k
:= range app.TaskInformation[stageNum] { // getting first taskNum when iterating to remove; randomized
because of GO taskNum = k break }

deletedTaskIp, exists := app.TaskInformation[stageNum][taskNum] if !exists { fmt.Printf("Failed to remove
task: %d, stage %d: not exists", taskNum, stageNum) return }

```

```

delete(app.TaskInformation[stageNum], taskNum) if stageNum == 0 && !app.DoneReading { temp := make(map[int]net.IP) for task, ip := range app.TaskInformation[0] { temp[task] = ip.Ip } app.Stage1UpdatesChan <- temp } app.sendIps()
task := Task{ TaskNumber: taskNum, Stage: stageNum, Executable: app.Ops[stageNum], } var reply int
rpcWorkersLock.RLock() rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()] rpcWorkersLock.RUnlock() err := rpcWorker.Call("Worker.AutoscaleDown", task, &reply) if err != nil { fmt.Println("Failed to send request to kill task: " + err.Error()) } }
func processStdin(i1 chan<- RainStorm) { scanner := bufio.NewScanner(os.Stdin) for { scanner.Scan() stdin := scanner.Text() stdin = strings.TrimSpace(stdin) splits := strings.Split(stdin, " ") switch splits[0] { case "RainStorm": // var rainStorm RainStorm bad := false for i := 1; i < len(splits) && !bad; i++ { var err error switch { case i == 1: // NumStages rainStorm.NumStages, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumStages: " + err.Error()) bad = true } break case i == 2: // NumTasksPerStage rainStorm.NumTasksPerStage, err = strconv.Atoi(splits[i]) if err != nil { fmt.Println("Failed to parse NumTasksPerStage: " + err.Error()) bad = true } break case 3 <= i && i < len(splits)-7: // processing operations and arguments rainStorm.Ops = append(rainStorm.Ops, Operation{Name: OperationName(splits[i]), Args: splits[i+1:]}) i++ break case i == len(splits)-7: // HydfsSrcDirectory rainStorm.HydfsSrcDirectory = splits[i] break case i == len(splits)-6: // HydfsDestinationFileName rainStorm.HydfsDestinationFileName = splits[i] break case i == len(splits)-5: // ExactlyOnce rainStorm.ExactlyOnce, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse ExactlyOnce: " + err.Error()) bad = true } break case i == len(splits)-4: // AutoScale rainStorm.AutoScale, err = strconv.ParseBool(splits[i]) if err != nil { fmt.Println("Failed to parse AutoScale: " + err.Error()) bad = true } break case i == len(splits)-3: // InputRate rainStorm.InputRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse InputRate: " + err.Error()) bad = true } break case i == len(splits)-2: // LowestRate rainStorm.LowestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse LowestRate: " + err.Error()) bad = true } break case i == len(splits)-1: // HighestRate rainStorm.HighestRate, err = strconv.ParseFloat(splits[i], 64) if err != nil { fmt.Println("Failed to parse HighestRate: " + err.Error()) bad = true } break } } if !bad { i1 <- rainStorm } break
case "kill_task": //@TODO: add implementation for this vm := splits[1] pid, _ := strconv.Atoi(splits[2])
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { done := false for taskNum, info := range stage { if vm == info.Ip.String() && pid == info.Pid { rpcWorkersLock.RLock() worker := rpcWorkers[info.Ip.String()] var reply int _ = worker.Call("Worker.KillTask", TaskID{ Task: taskNum, Stage: stageNum, }, &reply) rpcWorkersLock.RUnlock() done = true break } } if done { break } }
curApp.Lock.RUnlock() break
case "list_tasks": //@TODO print local log file for task curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { for _, info := range stage { fmt.Printf("%s %d %s\n", info.Ip.String(), info.Pid, curApp.Ops[stageNum]) } } curApp.Lock.RUnlock() break
} } }

```

## Gemini

Yes, there is a very specific and correct reason for the PID difference.

**A PID (Process ID) is unique to a single running instance of a process.** You cannot "revive" a dead PID.

When a task dies and you restart it, the Worker machine spins up a **brand new process**, which the Operating System assigns a **brand new PID**.

Here is the exact sequence occurring in your logs:

1. **Failure:** Task with PID 100 dies.
2. **Leader ReceiveFailure :**
  - Your code logs: "Restarting Task... PID: 100" (It prints the **Old PID** from your struct, confirming exactly which process died).
3. **Leader addTask :** calls the Worker.
4. **Worker:** Starts a new process. OS assigns PID 101 .

5. **Leader:** Updates struct to 101 .

#### 6. Leader addTask (End):

- Your code logs: "Starting Task... PID: 101" (It prints the **New PID**, confirming the restart was successful).

**Conclusion:** Seeing different PIDs is proof your system is working correctly.

## ⚠️ IMPORTANT: Fix for "System Getting Stuck"

In the code you pasted, you still have the logic that causes the system to hang when a failure occurs.

You are using **Blocking Sends** inside a **Lock**. If your Worker fails, your Sender loop (Input Reader) often hangs for a few seconds waiting for the network Timeout ( Dial or Write ). During this time, **it is not reading from the channel**.

If `ReceiveFailure` runs during this moment:

1. It grabs the Lock.
2. It tries to write to the channel.
3. The channel is full (or sender is busy).
4. **Leader Deadlocks:** `ReceiveFailure` waits forever. It holds the lock, so `ReceiveRateUpdate` stops working (Rates go to 0).

## The Solution

You must use **Non-Blocking Sends**. You do not need to worry about "dropping" updates. Since you send the **Full Map** every time, the Sender only needs the *latest* one. If the channel is full, skipping an update is actually an optimization (State Coalescing), not data loss.

**Change these 3 blocks in your code immediately:**

#### 1. In `ReceiveFailure`

```
Go □

// FIX: Remove "!app.DoneReading" check to ensure update always goes out
if task.Stage == 0 {
    temp := make(map[int]net.IP)
    for t, ip := range app.TaskInformation[0] {
        temp[t] = ip.Ip
    }
    // FIX: Use Select/Default to prevent Leader Deadlock
    select {
        case app.Stage1UpdatesChan <- temp:
        default:
            // Channel full. Safe to skip because the sender will
            // get the correct state in the next update anyway.
    }
}
```

#### 2. In `addTask`

Go



```
if stageNum == 0 {  
    temp := make(map[int]net.IP)  
    for task, ip := range app.TaskInformation[0] {  
        temp[task] = ip.Ip  
    }  
    // FIX: Use Select/Default  
    select {  
        case app.Stage1UpdatesChan <- temp:  
        default:  
    }  
}
```

### 3. In removeTask

Go



```
if stageNum == 0 {  
    temp := make(map[int]net.IP)  
    for task, ip := range app.TaskInformation[0] {  
        temp[task] = ip.Ip  
    }  
    // FIX: Use Select/Default  
    select {  
        case app.Stage1UpdatesChan <- temp:  
        default:  
    }  
}
```

## User

why could I be getting this error: Ack Read Error: read tcp 172.22.158.45:57610->172.22.94.231:8022: i/o timeout

## Gemini

The error `read tcp ... i/o timeout` means your Leader sent a tuple to the Worker, but the Worker **did not send a response back within 3 seconds**.

You set this deadline in your code here:

Go



```
const clientTimeout = time.Second * 3
// ...
_ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
ack, err := client.Buf.ReadString('\n') // <--- TIMEOUT HAPPENS HERE
```

Here are the three reasons why this is happening, ordered from most likely to least likely.

### 1. The Worker Logic is "Eating" the Tuple (Bug)

The Worker receives the tuple successfully, but it **forgets to write the ACK back**, or it writes the ACK incorrectly.

- **Check:** Does your Worker code have an explicit `conn.Write` or `fmt.Fprintf` sending the ACK?
- **Check:** Did you forget the **newline character** (`\n`)?
  - *Bad:* `fmt.Fprintf(conn, "temp-100-ACK")` (Leader waits forever for `\n`).
  - *Good:* `fmt.Fprintf(conn, "temp-100-ACK\n")` .
- **Check:** Are you flushing the buffer? If you use `bufio.NewWriter` on the worker, the ACK might be sitting in RAM and not on the wire.

### 2. The Worker is Blocked/Deadlocked

The Worker accepted the connection, but it is stuck waiting on something else before it can process the tuple and send the ACK.

- **Scenario:** The Worker is trying to write to a log file, but the log file lock is held by another goroutine.
- **Scenario:** The Worker is trying to send a message to *another node* (e.g., HyDFS or another stage), and *that connection* is blocked.
- **Debug:** Add a `fmt.Println("Worker received tuple, processing...")` at the very top of your Worker's tuple handler. If you don't see it, the Worker is deadlocked *accepting* the connection. If you see it but don't see "Sent ACK", the logic inside is stuck.

### 3. The Worker is Too Slow (Performance)

The Worker is working correctly, but processing the tuple takes longer than 3 seconds.

- **Cause:** Your `InputRate` is too high, and the Worker's CPU is pegged at 100%. It queues up the request and can't get to it in time.
- **Cause:** The Worker is performing a heavy disk operation (HyDFS append) for every single tuple.

- **Diagnostic:** Temporarily increase `clientTimeout` to `10 * time.Second`.
  - If it works: Your worker is just slow. You need to optimize (buffer writes) or lower the input rate.
  - If it still times out: Your worker is broken/deadlocked (Reason 1 or 2).

## Next Step

Look at your **Worker Code** (specifically the function handling `TuplePort` connections). Verify it ends with something like this:

Go



```
// Worker side
ackMsg := fmt.Sprintf("%s-%d-ACK\n", tupleId, seqNum) // Must match Leader's expected format
_, err := conn.Write([]byte(ackMsg))
if err != nil {
    fmt.Println("Failed to send ACK:", err)
}
```

**User**

ok but i do not want infinitely check for this.

**Gemini**

You are absolutely right. An infinite retry loop on the Leader is dangerous because if a worker is permanently broken (or the network partition is permanent), your entire application will hang forever on that one tuple.

You should implement a **Finite Retry Policy**. If a tuple fails to send after X attempts, you should drop it (or log it as a permanent failure) and move on to keep the stream alive.

## The Fix: Bounded Retries

Replace your Consumer Routine's inner `for` loop with this logic.

**Logic Changes:**

1. Added `maxRetries = 5`.
2. Increment `attempt` counter.
3. If `attempt > maxRetries`, we **break** the loop, log the failure, and move to the next tuple.

Go



```
// Consumer Routine
for {
    select {
    case <-ctx.Done():
        for _, c := range tupleClients { c.Conn.Close() }
        return

    case tuple := <-readingChan:
        if tuple.lineNum == -1 {
            r.Lock.Lock()
            r.DoneReading = true
            r.Lock.Unlock()

            r.sendStageCompletion(-1)
            for _, c := range tupleClients { c.Conn.Close() }
            return
        }

        // --- BOUNDED RETRY LOOP ---
        maxRetries := 5
        success := false

        for attempt := 0; attempt < maxRetries; attempt++ {
            // 1. Check for Topology Updates
            select {
            case updatedMap := <-r.Stage1UpdatesChan:
                firstStageTasks = updatedMap
```

```
firstTaskList = make([]int, 0)
for k := range firstStageTasks {
    firstTaskList = append(firstTaskList, k)
}
sort.Ints(firstTaskList)
default:
}

if len(firstTaskList) == 0 {
    time.Sleep(50 * time.Millisecond)
    continue
}

// 2. Routing & Connection
nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
nextTaskIp := firstStageTasks[nextTask]

client, ok := tupleClients[nextTaskIp.String()]
if !ok {
    conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
    if err != nil {
        fmt.Printf("Dial failed (attempt %d/%d): %v\n", attempt+1, maxRet
        time.Sleep(100 * time.Millisecond)
        continue
    }
    client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
    tupleClients[nextTaskIp.String()] = client
}

// 3. Send & Ack
var ack string
var sendErr, ackErr error

_ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
_, sendErr = fmt.Fprintf(client.Conn, "%s-%d,%d,%s\n", "temp",
tuple.lineNum)

if sendErr == nil {
    _ = client.Conn.SetReadDeadline(time.Now().Add(clientTimeout))
    ack, ackErr = client.Buf.ReadString('\n')
}

expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
isAckValid := (ackErr == nil) && (strings.TrimSpace(ack) == expectedAck)

if sendErr == nil && isAckValid {
    success = true
}
```

```
        break // SUCCESS! Exit retry loop
    }

    // 4. Failure Handling
    if sendErr != nil {
        fmt.Printf("Send Error (attempt %d): %v\n", attempt+1, sendErr)
    } else if ackErr != nil {
        fmt.Printf("Ack Error (attempt %d): %v\n", attempt+1, ackErr)
    }

    client.Conn.Close()
    delete(tupleClients, nextTaskIp.String())

    // Backoff before next attempt
    time.Sleep(200 * time.Millisecond)
}

if !success {
    // We failed 5 times. Drop tuple and move on to prevent system freeze.
    fmt.Printf("CRITICAL: Failed to send tuple %d after %d attempts. Dropping tuple.\n", attempt, attempt)
    r.LogErrorChan <- fmt.Sprintf("Data Loss: Tuple %d dropped after max retries.", attempt)
}

// Rate Limiting
numProcessed++
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.Second))
targetTime := startTime.Add(expectedDuration)
if targetTime.After(time.Now()) {
    time.Sleep(targetTime.Sub(time.Now()))
}
}
```

## Why this is safer

- **System Liveness:** If a worker goes into a "Zombie" state (accepts connections but never ACKs), your Leader won't get stuck forever on one line. It will try 5 times (taking ~15 seconds total) and then skip it.
- **Logging:** You explicitly log the data loss event so you can debug *why* it happened later without halting the entire application.

## User

wait but listen, the retry stuff will happen when a task failure occurs. so the line needs to be sent back to channel to process again after the ips have been updated

## Gemini

You are absolutely correct. Since a task failure is a temporary state that *will* be fixed (by the Leader updating the topology), you **must retry indefinitely** until the topology update arrives.

If you drop the tuple after 5 attempts, you lose data just because the Leader was a bit slow to restart the task. However, you should **not** send the line back to `readingChan` (`readingChan <- tuple`).

- **Why?** `readingChan` is a First-In-First-Out queue. If you push a failed tuple back, it goes to the **end of the line**, behind thousands of other tuples. This breaks the order of your stream (Tuple 1 gets processed after Tuple 5000).

## The Solution: Infinite Retry Loop with Topology Polling

You must stay inside the loop, retrying the *same* tuple forever, but **checking for IP updates** between every attempt.

1. **Attempt 1:** Send to IP `10.0.0.1`. **Fail** (Timeout).
2. **Action:** Close connection. Sleep 1s.
3. **Check Channel:** Did Leader send a new map? (No).
4. **Attempt 2:** Send to IP `10.0.0.1`. **Fail** (Timeout).
5. **Event:** Leader receives `ReceiveFailure` RPC and sends new map to channel.
6. **Check Channel:** Did Leader send a new map? **YES!** (New IP: `10.0.0.2`).
7. **Attempt 3:** Send to IP `10.0.0.2`. **Success!**

## The Code

Replace your "Consumer Routine" loop with this logic. It waits patiently for the fix without dropping data or breaking order.

Go

```
// Consumer Routine
for {
    select {
        case <-ctx.Done():
            for _, c := range tupleClients { c.Conn.Close() }
            return

        case tuple := <-readingChan:
            if tuple.lineNum == -1 {
                // Done Reading
                r.Lock.Lock()
                r.DoneReading = true
                r.Lock.Unlock()
                r.sendStageCompletion(-1)
            }
    }
}
```

```
        for _, c := range tupleClients { c.Conn.Close() }
        return
    }

    // --- INFINITE RETRY LOOP ---
    // We will stay here FOREVER until this specific tuple is sent.
    // This handles the "Waiting for Leader to fix Failure" scenario.
    for {
        // 1. ALWAYS check for Topology Updates first
        select {
            case updatedMap := <-r.Stage1UpdatesChan:
                firstStageTasks = updatedMap
                firstTaskList = make([]int, 0)
                for k := range firstStageTasks {
                    firstTaskList = append(firstTaskList, k)
                }
                sort.Ints(firstTaskList)
                // Clear broken connections on topology change to force redial
                for k, c := range tupleClients {
                    c.Conn.Close()
                    delete(tupleClients, k)
                }
            default:
        }

        // 2. Wait for valid topology
        if len(firstTaskList) == 0 {
            time.Sleep(100 * time.Millisecond)
            continue
        }

        // 3. Calculate Target (Hash or Round Robin)
        nextTask := firstTaskList[tuple.lineNum%len(firstTaskList)]
        nextTaskIp := firstStageTasks[nextTask]

        // 4. Get/Dial Client
        client, ok := tupleClients[nextTaskIp.String()]
        if !ok {
            conn, err := net.Dial("tcp", nextTaskIp.String()+TuplePort)
            if err != nil {
                fmt.Println("Dial failed, waiting for node to recover...", err)
                time.Sleep(500 * time.Millisecond) // Wait longer for recovery
                continue
            }
            client = &WorkerClient{Conn: conn, Buf: bufio.NewReader(conn)}
            tupleClients[nextTaskIp.String()] = client
        }
    }
}
```

```
        }
```

```
        // 5. Send & Expect ACK
        var ack string
        var sendErr, ackErr error

        _ = client.Conn.SetWriteDeadline(time.Now().Add(clientTimeout))
        _, sendErr = fmt.Fprintf(client.Conn, "%s-%d,%d,%s\n", "temp", tuple.lineNum, "ACK")
        ack, ackErr = client.Buf.ReadString('\n')
    }

    expectedAck := fmt.Sprintf("%s-%d-%s", "temp", tuple.lineNum, "ACK")
    isAckValid := (ackErr == nil) && (strings.TrimSpace(ack) == expectedAck)

    if sendErr == nil && isAckValid {
        // SUCCESS! Exit the retry loop and move to next tuple
        break
    }

    // 6. Failure handling
    // If we are here, the worker is dead or slow.
    // We DO NOT exit. We loop back and check for updates.
    fmt.Printf("Retrying tuple %d due to error: Send=%v Ack=%v\n", tuple.lineNum, sendErr, ack)
    client.Conn.Close()
    delete(tupleClients, nextTaskIp.String())

    // Important: Sleep to give Leader time to process the failure and send
    time.Sleep(1 * time.Second)
}

// 7. Rate Limiting (Only happens after success)
numProcessed++
expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(ti
targetTime := startTime.Add(expectedDuration)
if targetTime.After(time.Now()) {
    time.Sleep(targetTime.Sub(time.Now())))
}
}
```