# gemini-conversation

*Exported on: 12/7/2025, 10:29:14 PM*

---

**User**

hello , I am going to attach a leader below. There is also a worker that i am working on. for some reason when i kill a task the system stops streaming. why could this be

```
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 // [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 }
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.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) // 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)
localOutputFile.WriteString(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 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() }()
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, 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.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**

Based on the code provided, the reason your system stops streaming when you kill a task is that **the Leader never updates its state to reflect that the task is dead.**

## The Problem: The "Zombie" Task Loop

1. **State Mismatch:** When you run `kill_task`, you send an RPC to the worker to kill the process. However, the Leader's `TaskInformation` map still points to that old, dead IP address/PID.
2. **Input Reader Loop:** Look at your Input Reader logic (lines 362-385).
    - It picks a task from `firstStageTasks`.
    - If the task was killed, `net.Dial` or `client.Buf.ReadString` returns an error.
    - The code executes `delete(tupleClients, ...)` and `continue`.
    - **Crucially:** In the next iteration, `firstStageTasks` has not changed. It tries to connect to the **same dead task** again.
3. **Result:** The system enters an infinite retry loop trying to talk to the dead task. Since it can't move past that tuple, the stream halts.

## The Fix

You need to manually trigger the **Recovery** logic immediately after you kill the task. In a real distributed system, a failure detector would do this. Since this is a manual "Chaos Monkey" command, you should trigger `ReceiveFailure` manually within the `kill_task` block.

Here is the corrected `processStdin` function (specifically the `kill_task` case):

Go

```go
case "kill_task":
    vm := splits[1]
    pid, _ := strconv.Atoi(splits[2])

    // We need to release the lock before calling ReceiveFailure to avoid Deadlock,
    // because ReceiveFailure acquires the lock itself.
    var foundTask bool
    var targetTask Task

    curApp.Lock.RLock()
    for stageNum, stage := range curApp.TaskInformation {
       for taskNum, info := range stage {
          if vm == info.Ip.String() && pid == info.Pid {
             targetTask = Task{
                 TaskNumber: taskNum,
                 Stage:      stageNum,
             }
             foundTask = true
             break
          }
       }
       if foundTask {
          break
       }
    }
    curApp.Lock.RUnlock()

    if foundTask {
        // 1. Kill the physical process on the worker
        rpcWorkersLock.RLock()
        worker, ok := rpcWorkers[vm] // Use the vm string to find the client
        rpcWorkersLock.RUnlock()

        if ok {
            var reply int
            err := worker.Call("Worker.KillTask", TaskID{
                Task:  targetTask.TaskNumber,
                Stage: targetTask.Stage,
            }, &reply)

            if err != nil {
                fmt.Println("Error killing task:", err)
            } else {
                fmt.Println("Task Killed via RPC.")
            }
        }

        // 2. IMPORTANT: Tell the Leader to recover/reschedule immediately
        fmt.Println("Triggering Leader Recovery...")
        var reply int
        // We call ReceiveFailure locally to update the map and send new IPs to everyone
        curApp.ReceiveFailure(targetTask, &reply)
    } else {
        fmt.Println("Task not found to kill.")
    }
    break
```

### Why this fixes it

1. **`curApp.ReceiveFailure`** (Lines 434-455 in your code) is called.
2. It assigns a **new worker IP** to that `taskNum`.
3. It calls `app.sendIps()`, which broadcasts the updated `TaskInformation` map to all workers.
4. It pushes the update to `Stage1UpdatesChan`.
5. Your **Input Reader** (Line 342) receives the update from the channel, updates its local map, and stops trying to dial the dead IP, allowing the stream to continue.

### Important Note on Locks

I modified the locking logic in the fix above. In your original code, you held `curApp.Lock.RLock()` while potentially needing to call `ReceiveFailure`.

- `ReceiveFailure` performs `app.Lock.Lock()`.
- If you hold `RLock` and try to call a function that needs `Lock`, you might cause a deadlock depending on how `ReceiveFailure` is invoked (locally vs RPC).
- **Best Practice:** Find the task info, release the lock, *then* perform the Kill/Recover actions.

**User**

would the worker still call the RecieveFailure function?

**Gemini**

It depends entirely on how you implemented your **Worker** code (which isn't shown here), but based on the symptom you described ("the system stops streaming"), the answer is almost certainly **No**.

Here is why relying on the Worker to call `ReceiveFailure` is causing your issue:

### 1. The "Fire and Forget" Problem

In your current `kill_task` implementation, the Leader sends a command to kill the process and then immediately moves on. It assumes nothing else needs to be done.

- **If your Worker code looks like this:**

  Go

  ```go
  func (w *Worker) KillTask(...) error {
      // Find process
      process.Kill()
      return nil
  }
  ```

  Then the Worker simply kills the process and forgets about it. It does **not** notify the Leader. The Leader still thinks the task is alive at the old IP/PID, so it keeps sending data there, resulting in the stream halting.

### 2. The Race Condition

Even if your Worker has a monitoring function (like `cmd.Wait()`) that detects the crash and calls `ReceiveFailure`, there is a **race condition**.

1. Leader sends `kill_task`.
2. Leader Input Reader tries to send the next tuple to the task *immediately*.
3. Worker kills the task.
4. Worker detects death and calls `ReceiveFailure` (this takes network time).

During that gap between step 2 and step 4, your Input Reader hits a dead end.

### 3. The Logical Fix

Since **the Leader** is the one initiating the `kill_task` command, the Leader **already knows** the task is about to die. It shouldn't wait for the Worker to tell it news it already knows.

**You should update the state on the Leader immediately.**

Here is the updated logic for your `kill_task` block to ensure `ReceiveFailure` is called regardless of what the Worker does:

Go

```
case "kill_task":
    vm := splits[1]
    pid, _ := strconv.Atoi(splits[2])

    // 1. Find the task info safely
    var targetTask Task
    found := false

    curApp.Lock.RLock()
    for sNum, stage := range curApp.TaskInformation {
       for tNum, info := range stage {
          if vm == info.Ip.String() && pid == info.Pid {
             targetTask = Task{Stage: sNum, TaskNumber: tNum}
             found = true
             break
          }
       }
       if found { break }
    }
    curApp.Lock.RUnlock()

    if found {
       // 2. Kill the physical process
       rpcWorkersLock.RLock()
       client := rpcWorkers[vm]
       rpcWorkersLock.RUnlock()

       var reply int
       _ = client.Call("Worker.KillTask", TaskID{Stage: targetTask.Stage, Task: targetTask.

       // 3. FORCE the state update immediately
       fmt.Println("Leader initiating recovery for killed task...")
       curApp.ReceiveFailure(targetTask, &reply)
    }
    break
```

**Why this works:** By calling `curApp.ReceiveFailure` manually here, you force the Leader to:

1. Mark the old IP as dead.
2. Pick a new IP.
3. Broadcast the new IP to all other workers.
4. **Update the Input Reader** so it stops trying to dial the dead task and dials the new one instead.

**User**

I dumped the goroutines when I ran it with this and this is what I got: Goroutine 1 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.chanrecv (/usr/lib/golang/src/runtime/chan.go:667) runtime.chanrecv1 (/usr/lib/golang/src/runtime/chan.go:509) main.main (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:399) runtime.main (/usr/lib/golang/src/runtime/proc.go:285) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: runtime.newproc (<autogenerated>:1)
Goroutine 2 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.goparkunlock (/usr/lib/golang/src/runtime/proc.go:466) runtime.forcegchelper (/usr/lib/golang/src/runtime/proc.go:373) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: runtime.init.7 (/usr/lib/golang/src/runtime/proc.go:361)
Goroutine 5 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.netpollblock (/usr/lib/golang/src/runtime/netpoll.go:575) internal/poll.runtime_pollWait (/usr/lib/golang/src/runtime/netpoll.go:351) internal/poll.(*pollDesc).wait (/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84) internal/poll.(*pollDesc).waitRead (/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89) internal/poll.(*FD).Accept (/usr/lib/golang/src/internal/poll/fd_unix.go:613) net.(*netFD).accept (/usr/lib/golang/src/net/fd_unix.go:161) net.(*TCPListener).accept (/usr/lib/golang/src/net/tcpsock_posix.go:159) net.(*TCPListener).Accept (/usr/lib/golang/src/net/tcpsock.go:380) main.main.func3 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:150) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.main (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:146)
Goroutine 6 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.chanrecv (/usr/lib/golang/src/runtime/chan.go:667) runtime.chanrecv1 (/usr/lib/golang/src/runtime/chan.go:509) main.main.func3.1 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:148) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.main.func3 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:148)
Goroutine 7 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.netpollblock (/usr/lib/golang/src/runtime/netpoll.go:575) internal/poll.runtime_pollWait (/usr/lib/golang/src/runtime/netpoll.go:351) internal/poll.(*pollDesc).wait (/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84) internal/poll.(*pollDesc).waitRead (/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89) internal/poll.(*FD).Read (/usr/lib/golang/src/internal/poll/fd_unix.go:165) net.(*netFD).Read (/usr/lib/golang/src/net/fd_posix.go:68) net.(*conn).Read (/usr/lib/golang/src/net/net.go:196) bufio.(*Reader).Read (/usr/lib/golang/src/bufio/bufio.go:245) io.ReadAtLeast (/usr/lib/golang/src/io/io.go:335) io.ReadFull (/usr/lib/golang/src/io/io.go:354) encoding/gob.decodeUintReader (/usr/lib/golang/src/encoding/gob/decode.go:116) encoding/gob.(*Decoder).recvMessage (/usr/lib/golang/src/encoding/gob/decoder.go:84) encoding/gob.(*Decoder).decodeTypeSequence (/usr/lib/golang/src/encoding/gob/decoder.go:150) encoding/gob.(*Decoder).DecodeValue (/usr/lib/golang/src/encoding/gob/decoder.go:229) encoding/gob.(*Decoder).Decode (/usr/lib/golang/src/encoding/gob/decoder.go:206) net/rpc.(*gobClientCodec).ReadResponseHeader (/usr/lib/golang/src/net/rpc/client.go:228) net/rpc.(*Client).input (/usr/lib/golang/src/net/rpc/client.go:109) net/rpc.NewClientWithCodec.gowrap1 (/usr/lib/golang/src/net/rpc/client.go:206) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: net/rpc.NewClientWithCodec (/usr/lib/golang/src/net/rpc/client.go:206)
Goroutine 8 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.chanrecv (/usr/lib/golang/src/runtime/chan.go:667) runtime.chanrecv1 (/usr/lib/golang/src/runtime/chan.go:509) main.main.func5.1 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:242) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.main.func5 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:241)
Goroutine 9 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.netpollblock

```
(/usr/lib/golang/src/runtime/netpoll.go:575)   internal/poll.runtime_pollWait
(/usr/lib/golang/src/runtime/netpoll.go:351)   internal/poll.(*pollDesc).wait
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84)   internal/poll.(*pollDesc).waitRead
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89)   internal/poll.(*FD).Read
(/usr/lib/golang/src/internal/poll/fd_unix.go:165)   net.(*netFD).Read (/usr/lib/golang/src/net/fd_posix.go:68)
net.(*conn).Read (/usr/lib/golang/src/net/net.go:196)   bufio.(*Reader).fill
(/usr/lib/golang/src/bufio/bufio.go:113)   bufio.(*Reader).ReadSlice (/usr/lib/golang/src/bufio/bufio.go:380)
bufio.(*Reader).collectFragments (/usr/lib/golang/src/bufio/bufio.go:455)   bufio.(*Reader).ReadString
(/usr/lib/golang/src/bufio/bufio.go:502)   main.main.func5.2 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:262)   main.main.func5.gowrap2 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:275)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main.func5 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:253)
Goroutine 10   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.chanrecv
(/usr/lib/golang/src/runtime/chan.go:667)   runtime.chanrecv1 (/usr/lib/golang/src/runtime/chan.go:509)
main.main.func5.2.1 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:258)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main.func5.2 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:258)
Goroutine 18   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.goparkunlock
(/usr/lib/golang/src/runtime/proc.go:466)   runtime.bgsweep (/usr/lib/golang/src/runtime/mgcsweep.go:279)
runtime.gcenable.gowrap1 (/usr/lib/golang/src/runtime/mgc.go:212)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: runtime.gcenable
(/usr/lib/golang/src/runtime/mgc.go:212)
Goroutine 19   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.goparkunlock
(/usr/lib/golang/src/runtime/proc.go:466)   runtime.(*scavengerState).park
(/usr/lib/golang/src/runtime/mgcscavenge.go:425)   runtime.bgscavenge
(/usr/lib/golang/src/runtime/mgcscavenge.go:653)   runtime.gcenable.gowrap2
(/usr/lib/golang/src/runtime/mgc.go:213)   runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)
created at: runtime.gcenable (/usr/lib/golang/src/runtime/mgc.go:213)
Goroutine 20   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.runFinalizers
(/usr/lib/golang/src/runtime/mfinal.go:210)   runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)
created at: runtime.createfing (/usr/lib/golang/src/runtime/mfinal.go:172)
Goroutine 21   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.goparkunlock
(/usr/lib/golang/src/runtime/proc.go:466)   runtime.(*cleanupQueue).dequeue
(/usr/lib/golang/src/runtime/mcleanup.go:439)   runtime.runCleanups
(/usr/lib/golang/src/runtime/mcleanup.go:635)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: runtime.(*cleanupQueue).createGs
(/usr/lib/golang/src/runtime/mcleanup.go:589)
Goroutine 22   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.netpollblock
(/usr/lib/golang/src/runtime/netpoll.go:575)   internal/poll.runtime_pollWait
(/usr/lib/golang/src/runtime/netpoll.go:351)   internal/poll.(*pollDesc).wait
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84)   internal/poll.(*pollDesc).waitRead
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89)   internal/poll.(*FD).Accept
(/usr/lib/golang/src/internal/poll/fd_unix.go:613)   net.(*netFD).accept (/usr/lib/golang/src/net/fd_unix.go:161)
net.(*TCPListener).accept (/usr/lib/golang/src/net/tcpsock_posix.go:159)   net.(*TCPListener).Accept
(/usr/lib/golang/src/net/tcpsock.go:380)   net/rpc.(*Server).Accept (/usr/lib/golang/src/net/rpc/server.go:631)
net/rpc.Accept (/usr/lib/golang/src/net/rpc/server.go:691)   main.main.func1 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:73)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:66)
Goroutine 23   internal/runtime/syscall.Syscall6
(/usr/lib/golang/src/internal/runtime/syscall/asm_linux_amd64.s:36)   syscall.RawSyscall6
```

```
(/usr/lib/golang/src/syscall/syscall_linux.go:65)   syscall.Syscall (/usr/lib/golang/src/syscall/syscall_linux.go:87)
  syscall.read (/usr/lib/golang/src/syscall/zsyscall_linux_amd64.go:736)   syscall.Read
(/usr/lib/golang/src/syscall/syscall_unix.go:183)   internal/poll.ignoringEINTRIO
(/usr/lib/golang/src/internal/poll/fd_unix.go:738)   internal/poll.(*FD).Read
(/usr/lib/golang/src/internal/poll/fd_unix.go:161)   os.(*File).read (/usr/lib/golang/src/os/file_posix.go:29)   os.
(*File).Read (/usr/lib/golang/src/os/file.go:144)   bufio.(*Scanner).Scan (/usr/lib/golang/src/bufio/scan.go:219)
main.processStdin (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:652)
main.main.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:77)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:77)
Goroutine 28   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.netpollblock
(/usr/lib/golang/src/runtime/netpoll.go:575)   internal/poll.runtime_pollWait
(/usr/lib/golang/src/runtime/netpoll.go:351)   internal/poll.(*pollDesc).wait
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84)   internal/poll.(*pollDesc).waitRead
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89)   internal/poll.(*FD).Read
(/usr/lib/golang/src/internal/poll/fd_unix.go:165)   net.(*netFD).Read (/usr/lib/golang/src/net/fd_posix.go:68)
net.(*conn).Read (/usr/lib/golang/src/net/net.go:196)   bufio.(*Reader).Read
(/usr/lib/golang/src/bufio/bufio.go:245)   io.ReadAtLeast (/usr/lib/golang/src/io/io.go:335)   io.ReadFull
(/usr/lib/golang/src/io/io.go:354)   encoding/gob.decodeUintReader
(/usr/lib/golang/src/encoding/gob/decode.go:116)   encoding/gob.(*Decoder).recvMessage
(/usr/lib/golang/src/encoding/gob/decoder.go:84)   encoding/gob.(*Decoder).decodeTypeSequence
(/usr/lib/golang/src/encoding/gob/decoder.go:150)   encoding/gob.(*Decoder).DecodeValue
(/usr/lib/golang/src/encoding/gob/decoder.go:229)   encoding/gob.(*Decoder).Decode
(/usr/lib/golang/src/encoding/gob/decoder.go:206)   net/rpc.(*gobClientCodec).ReadResponseHeader
(/usr/lib/golang/src/net/rpc/client.go:228)   net/rpc.(*Client).input (/usr/lib/golang/src/net/rpc/client.go:109)
net/rpc.NewClientWithCodec.gowrap1 (/usr/lib/golang/src/net/rpc/client.go:206)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: net/rpc.NewClientWithCodec
(/usr/lib/golang/src/net/rpc/client.go:206)
Goroutine 29   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.selectgo
(/usr/lib/golang/src/runtime/select.go:351)   main.main.func2 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:119)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:106)
Goroutine 30   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.netpollblock
(/usr/lib/golang/src/runtime/netpoll.go:575)   internal/poll.runtime_pollWait
(/usr/lib/golang/src/runtime/netpoll.go:351)   internal/poll.(*pollDesc).wait
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84)   internal/poll.(*pollDesc).waitRead
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89)   internal/poll.(*FD).Read
(/usr/lib/golang/src/internal/poll/fd_unix.go:165)   net.(*netFD).Read (/usr/lib/golang/src/net/fd_posix.go:68)
net.(*conn).Read (/usr/lib/golang/src/net/net.go:196)   bufio.(*Reader).Read
(/usr/lib/golang/src/bufio/bufio.go:245)   io.ReadAtLeast (/usr/lib/golang/src/io/io.go:335)   io.ReadFull
(/usr/lib/golang/src/io/io.go:354)   encoding/gob.decodeUintReader
(/usr/lib/golang/src/encoding/gob/decode.go:116)   encoding/gob.(*Decoder).recvMessage
(/usr/lib/golang/src/encoding/gob/decoder.go:84)   encoding/gob.(*Decoder).decodeTypeSequence
(/usr/lib/golang/src/encoding/gob/decoder.go:150)   encoding/gob.(*Decoder).DecodeValue
(/usr/lib/golang/src/encoding/gob/decoder.go:229)   encoding/gob.(*Decoder).Decode
(/usr/lib/golang/src/encoding/gob/decoder.go:206)   net/rpc.(*gobServerCodec).ReadRequestHeader
(/usr/lib/golang/src/net/rpc/server.go:403)   net/rpc.(*Server).readRequestHeader
(/usr/lib/golang/src/net/rpc/server.go:589)   net/rpc.(*Server).readRequest
(/usr/lib/golang/src/net/rpc/server.go:549)   net/rpc.(*Server).ServeCodec
(/usr/lib/golang/src/net/rpc/server.go:464)   net/rpc.(*Server).ServeConn
(/usr/lib/golang/src/net/rpc/server.go:455)   main.main.func3.gowrap1 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:154)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main.func3 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:154)
Goroutine 31   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.chanrecv
(/usr/lib/golang/src/runtime/chan.go:667)   runtime.chanrecv2 (/usr/lib/golang/src/runtime/chan.go:514)
main.main.func4 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:224)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:209)
Goroutine 32   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.netpollblock
(/usr/lib/golang/src/runtime/netpoll.go:575)   internal/poll.runtime_pollWait
(/usr/lib/golang/src/runtime/netpoll.go:351)   internal/poll.(*pollDesc).wait
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:84)   internal/poll.(*pollDesc).waitRead
(/usr/lib/golang/src/internal/poll/fd_poll_runtime.go:89)   internal/poll.(*FD).Accept
(/usr/lib/golang/src/internal/poll/fd_unix.go:613)   net.(*netFD).accept (/usr/lib/golang/src/net/fd_unix.go:161)
net.(*TCPListener).accept (/usr/lib/golang/src/net/tcpsock_posix.go:159)   net.(*TCPListener).Accept
(/usr/lib/golang/src/net/tcpsock.go:380)   main.main.func5 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:247)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:238)
Goroutine 33   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.chansend
(/usr/lib/golang/src/runtime/chan.go:283)   runtime.chansend1 (/usr/lib/golang/src/runtime/chan.go:161)
main.main.func6 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:384)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:281)
Goroutine 34   runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)   runtime.chansend
(/usr/lib/golang/src/runtime/chan.go:283)   runtime.chansend1 (/usr/lib/golang/src/runtime/chan.go:161)
main.main.func6.1 (/home/manandp2/g14-mp4/RainStorm/RainStormLeader/main.go:309)   runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)   created at: main.main.func6 (/home/manandp2/g14-
mp4/RainStorm/RainStormLeader/main.go:306)
```

**Gemini**

Based on the stack dump, **Goroutine 33** and **Goroutine 34** identify the exact cause of your freeze.

## The Diagnosis: Channel Deadlock (The "Full Mailbox" Problem)

Your system is deadlocked because the **Input Reader** is trying to put a failed message back into a channel that is already full, but it can't empty the channel because it's blocked trying to fill it.

**1. Goroutine 33 (Input Reader) is stuck here:**

Go

```go
// main.go:384
readingChan <- tuple
```

This corresponds to your retry logic. When a task is killed, the send fails, and your code tries to push the tuple *back* into `readingChan` to retry it later.

**2. Goroutine 34 (File Scanner) is stuck here:**

Go

```
// main.go:309
readingChan <- struct{...}
```

This is the background thread reading from the input file. It is incredibly fast and has likely filled `readingChan` (capacity 100) completely while the main loop was busy trying to dial the worker.

**3. The Deadlock Scenario:**

1. `readingChan` is full (100/100 items).
2. The Input Reader (Goroutine 33) takes **one** item out (99/100).
3. The File Scanner (Goroutine 34) instantly sees a free spot and puts a new line in (100/100).
4. The Input Reader tries to process the item, fails (because the task is dead), and tries to execute `readingChan <- tuple`.
5. **BLOCK:** The channel is full (100/100). The Input Reader waits for space.
6. **DEADLOCK:** The only thing that can make space is the Input Reader itself (by reading the next item), but it can't read because it is stuck writing.

## The Fix

Do not push failed tuples back into the channel. Instead, keep the failed tuple in a local variable and retry it within the loop.

**Modify your Input Reader loop (inside `main.go`) as follows:**

Go

```go
// ... inside the Input Reader goroutine ...

// 1. Create a variable to hold a tuple that needs retrying
var pendingTuple *struct {
    line    string
    lineNum int
}

for {
    // 2. Determine what to process: either the pending retry or a new one from the channel
    var tuple struct {
        line    string
        lineNum int
    }

    if pendingTuple != nil {
        // We have a retry! Use it.
        tuple = *pendingTuple
        pendingTuple = nil // Clear it for now
    } else {
        // No retry pending, read from channel
        select {
```

```
        case <-ctx.Done():
            // ... cleanup ...
            return
        case t := <-readingChan:
            tuple = t
        }
    }

    if tuple.lineNum == -1 {
        // ... handle completion ...
        return
    }

    // ... [Get IP logic] ...

    // ... [Dialing logic] ...

    // ... [Sending logic] ...

    if err != nil || strings.TrimSpace(ack) != expectedAck {
        client.Conn.Close()
        delete(tupleClients, nextTaskIp.String())

        // FIX: DO NOT send back to channel.
        // readingChan <- tuple  <-- DELETE THIS LINE

        // INSTEAD: Store it locally to retry on the next loop iteration
        pendingTuple = &tuple

        // Optional: Add a small sleep to prevent CPU spinning while waiting for the task t
        time.Sleep(100 * time.Millisecond)
        continue
    }

    // Success!
    numProcessed++
    // ... [Rate limiting logic] ...
}
```

## Summary of Changes

1. **Remove `readingChan <- tuple`**: This eliminates the deadlock risk.
2. **Add `pendingTuple`**: This stores the failed item locally in memory.
3. **Modify Loop Logic**: At the start of the loop, check if `pendingTuple` exists. If yes, process it *instead* of reading from the channel.

**User**

is this correct:

```
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 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() }()
var pendingTuple *struct { line string lineNum int }
for { var tuple struct { line string lineNum int } if pendingTuple != nil { // We have a retry! Use it. tuple = *pendingTuple pendingTuple = nil // Clear it for now } else { select { case <-ctx.Done(): for _, c := range tupleClients { c.Conn.Close() } return case t := <-readingChan: tuple := t } }
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, 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 pendingTuple = &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())) } } }()
```

**Gemini**

You are very close, but there are **three critical errors** in this snippet that will cause data loss or compilation issues.

### 1. The Variable Shadowing Bug (Critical)

Inside your `else` block, you wrote:

Go

```go
case t := <-readingChan:
   tuple := t  // <--- ERROR: := creates a NEW variable only for this scope
```

By using `:=`, you are creating a *new* variable named `tuple` that only exists inside that `case` statement. The outer `tuple` variable (defined at the top of the loop) remains empty/zero-valued. When the code exits the `select`, it proceeds with an empty tuple.

**Fix:** Change `tuple := t` to `tuple = t`.

### 2. The Deadlock is still there (in `len == 0` check)

You correctly removed the push-back in the main failure case, but you left it here:

Go

```go
if len(firstTaskList) == 0 {
   time.Sleep(50 * time.Millisecond)
   readingChan <- tuple // <--- ERROR: Still causes deadlock if channel is full!
   continue
}
```

If the task list is empty, you push back to the channel. If the channel is full (which caused your original crash), you deadlock here.

**Fix:** Use `pendingTuple = &tuple` here as well.

### 3. Data Loss on Connection Failure

In your `net.Dial` error handling:

Go

```go
if err != nil {
   delete(tupleClients, nextTaskIp.String())
   continue // <--- ERROR: You forgot to save the tuple!
}
```

If `net.Dial` fails, you `continue` the loop. `pendingTuple` is currently `nil`. The loop restarts, sees `pendingTuple` is nil, and reads the **next** item from the channel. The current `tuple` is lost forever.

**Fix:** You must set `pendingTuple = &tuple` before continuing here too.

---

### Corrected Code

Here is the corrected version with those three fixes applied:

Go

```go
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
   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()
}()

// Tracks a tuple we failed to send and need to retry
var pendingTuple *struct {
   line    string
   lineNum int
}

for {
   // Definition of the tuple variable for this iteration
```

```
      var tuple struct {
         line    string
         lineNum int
      }

      // --- SELECTION LOGIC ---
      if pendingTuple != nil {
         // 1. Prioritize the retry
         tuple = *pendingTuple
         pendingTuple = nil
      } else {
         // 2. Otherwise read from channel
         select {
         case <-ctx.Done():
            for _, c := range tupleClients {
               c.Conn.Close()
            }
            return
         case t := <-readingChan:
            tuple = t // FIX 1: use '=' not ':='
         }
      }

      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 2: Handle empty task list without deadlocking channel
      if len(firstTaskList) == 0 {
         time.Sleep(50 * time.Millisecond)
```

```
			pendingTuple = &tuple // Save state
			continue              // Retry loop
		}

		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())
				// FIX 3: Save tuple before continuing on dial fail
				pendingTuple = &tuple
				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, nextTa

		_ = 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())

			// Retry logic (already correct in your snippet, kept for clarity)
			pendingTuple = &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()))
		}
	}
}()
```

**User**

I have pasted my updated leader code below. When i try to run the program with auto scaling all the task are created and immediately marked as completed in the log file and the application says it has completed :

```
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 // [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 }
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.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) // 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)
localOutputFile.WriteString(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 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() }()
// Tracks a tuple we failed to send and need to retry var pendingTuple *struct { line string lineNum int }
for { // Definition of the tuple variable for this iteration var tuple struct { line string lineNum int }
// --- SELECTION LOGIC --- if pendingTuple != nil { // 1. Prioritize the retry tuple = *pendingTuple pendingTuple
= nil } else { // 2. Otherwise read from channel select { case <-ctx.Done(): for _, c := range tupleClients {
c.Conn.Close() } return case t := <-readingChan: tuple = t // FIX 1: use '=' not ':=' } }
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 2: Handle empty task list without deadlocking channel if len(firstTaskList) == 0 { time.Sleep(50 *
time.Millisecond) pendingTuple = &tuple // Save state continue // Retry loop }
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()) // FIX 3: Save tuple before continuing on dial fail
pendingTuple = &tuple 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())
// Retry logic (already correct in your snippet, kept for clarity) pendingTuple = &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.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": //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 vm := splits[1] pid, _ := strconv.Atoi(splits[2])
// We need to release the lock before calling ReceiveFailure to avoid Deadlock, // because ReceiveFailure
acquires the lock itself. var foundTask bool var targetTask Task
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { for taskNum, info := range stage {
if vm == info.Ip.String() && pid == info.Pid { targetTask = Task{ TaskNumber: taskNum, Stage: stageNum, }
foundTask = true break } } if foundTask { break } } curApp.Lock.RUnlock()
if foundTask { // 1. Kill the physical process on the worker rpcWorkersLock.RLock() worker, ok :=
rpcWorkers[vm] // Use the vm string to find the client rpcWorkersLock.RUnlock()
if ok { var reply int err := worker.Call("Worker.KillTask", TaskID{ Task: targetTask.TaskNumber, Stage:
targetTask.Stage, }, &reply)
if err != nil { fmt.Println("Error killing task:", err) } else { fmt.Println("Task Killed via RPC.") } }
// 2. IMPORTANT: Tell the Leader to recover/reschedule immediately fmt.Println("Triggering Leader
Recovery...") var reply int // We call ReceiveFailure locally to update the map and send new IPs to everyone
curApp.ReceiveFailure(targetTask, &reply) } else { fmt.Println("Task not found to kill.") } 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**

The reason your application creates tasks and then immediately kills them (resulting in completion) is that **your Auto-scaling logic is inverted.**

## The Logic Error

In a standard distributed system:

1. **Rate > HighestRate:** This usually means the worker is **overloaded** (high CPU, large queue, or processing too many items). You need to **ADD** a worker (Upscale) to share the load.
2. **Rate < LowestRate:** This usually means the worker is **underutilized** (idle). You should **REMOVE** a worker (Downscale) to save resources.

**Your Code currently does the opposite:**

- It sees a High Rate and calls `removeTask` (killing a busy worker).
- It sees a Low Rate and calls `addTask`.

Because your workers likely start processing immediately (reporting a rate > 0 or > HighestRate), the leader interprets this as a signal to kill them. It aggressively downscales until the tasks are gone or the system becomes unstable and terminates.

### The Fix

You need to swap the logic inside `ReceiveRateUpdate`.

**Change this block (Lines 470-488):**

Go

```go
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
    app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n", args.Rate, args.Task

    if app.AutoScale {
       // CASE 1: Rate is LOW (Underutilized) -> We should DOWNSCALE (Remove Task)
       if args.Rate < app.LowestRate {
          app.Lock.Lock()
          // Only remove if we have more than 1 task
          if len(app.TaskInformation[args.Stage]) > 1 {
             app.LogFileChan <- fmt.Sprintf("Downscaling (Low Rate) Stage: %d Rate: %.2f\n"
             app.removeTask(args.Stage)
          }
          app.Lock.Unlock()

       // CASE 2: Rate is HIGH (Overloaded) -> We should UPSCALE (Add Task)
       } else if args.Rate > app.HighestRate {
          app.Lock.Lock()
          taskNum := app.NextTaskNum[args.Stage]
          app.NextTaskNum[args.Stage]++
          app.LogFileChan <- fmt.Sprintf("Upscaling (High Rate) Stage: %d Rate: %.2f\n", ar
          app.addTask(args.Stage, taskNum)
          app.sendIps()
          app.Lock.Unlock()
       }
    }
    return nil
}
```

### Summary of Changes

1. **Swapped the Conditions:**
    - `args.Rate < app.LowestRate` now triggers **Downscaling** (`removeTask`).
    - `args.Rate > app.HighestRate` now triggers **Upscaling** (`addTask`).
2. **Added Safety Check:** I added `if len(app.TaskInformation[args.Stage]) > 1` directly in the handler before acquiring the lock for `removeTask` to prevent spamming removal requests when you are already at

the minimum.

**User**

that is a good catch. however, here is the log file when i tried to run it: 2025-12-07 20:20:24: Started RainStorm Application 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.231 PID: 20065 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.46 PID: 31657 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.46 PID: 31926 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.232 PID: 32602 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.47 PID: 32849 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.47 PID: 32791 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.233 PID: 32840 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.48 PID: 32519 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.48 PID: 31036 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.231 PID: 20070 op_exe: Filter 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.46 PID: 31661 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.46 PID: 31931 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.232 PID: 32607 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.47 PID: 32853 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.47 PID: 32795 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.233 PID: 32844 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.48 PID: 32524 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.158.48 PID: 31041 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.94.231 PID: 20075 op_exe: Transform 2025-12-07 20:20:24: Starting Task at VM: 172.22.154.46 PID: 31667 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 5 Stage: 0 VM: 172.22.158.47 PID: 824634747032 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 7 Stage: 0 VM: 172.22.154.48 PID: 824639482536 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 4 Stage: 0 VM: 172.22.154.47 PID: 824639482616 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 2 Stage: 0 VM: 172.22.158.46 PID: 824639482696 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 6 Stage: 0 VM: 172.22.94.233 PID: 824639482776 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 3 Stage: 0 VM: 172.22.94.232 PID: 824634748584 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 8 Stage: 0 VM: 172.22.158.48 PID: 824639482872 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 1 Stage: 0 VM: 172.22.154.46 PID: 824639482952 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 9 Stage: 0 VM: 172.22.94.231 PID: 824639483032 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 0 Stage: 0 VM: 172.22.94.231 PID: 824634750040 op_exe: Filter 2025-12-07 20:20:24: Task Completed TaskID: 4 Stage: 1 VM: 172.22.158.47 PID: 824639487768 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 2 Stage: 1 VM: 172.22.94.232 PID: 824639487800 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 7 Stage: 1 VM: 172.22.158.48 PID: 824639487832 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 3 Stage: 1 VM: 172.22.154.47 PID: 824639487864 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 5 Stage: 1 VM: 172.22.94.233 PID: 824639487896 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 1 Stage: 1 VM: 172.22.158.46 PID: 824639487928 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 0 Stage: 1 VM: 172.22.154.46 PID: 824639487944 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 8 Stage: 1 VM: 172.22.94.231 PID: 824634901432 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 6 Stage: 1 VM: 172.22.154.48 PID: 824634901464 op_exe: Transform 2025-12-07 20:20:24: Task Completed TaskID: 9 Stage: 1 VM: 172.22.154.46 PID: 824634901504 op_exe: Transform 2025-12-07 20:20:24: RainStorm Application Completed

as you can see, the autoscaling was never hit as the rate was not even printed

**Gemini**

The logs indicate that your application is finishing **instantly** (in the same second it started). It starts the tasks, and then immediately shuts them down.

This is not an issue with autoscaling. **The issue is that your Input Reader is not reading any data.**

Here is the chain of events causing this:

1. **Input Reader Starts:** It tries to open the file at `filepath.Join(dataDir, r.HydfsSrcDirectory)`.

2. **Scan Fails:** The `scanner.Scan()` loop returns `false` immediately (either because the file wasn't found, is a directory, or is empty).
3. **EOS Sent:** The Input Reader sends the `-1` tuple (End of Stream) immediately.
4. **Shutdown Sequence:** The system broadcasts "Stage Completed" to all workers.
5. **Termination:** The workers finish, the Leader sees all tasks complete, and the app exits.

Because this happens instantly, the `numProcessed` count never increases, so the "Rate" calculation never happens, and thus no autoscaling logs are ever generated.

### The Fix

You need to fix the error handling where you open the file. Currently, you print the error but **allow the program to continue** with a bad file handle.

Modify the **Input Reader** section (around line 360 in your code) to panic or return if the file cannot be opened, and check for scanner errors.

Go

```go
// --- INPUT READER (PRODUCER 2) ---
wgProducers.Add(1)
go func() {
   defer wgProducers.Done()
   // Move the Open logic INSIDE the goroutine or check it strictly before
   fullPath := filepath.Join(dataDir, r.HydfsSrcDirectory)
   inputFile, err := os.Open(fullPath)
   if err != nil {
       // CRITICAL FIX: Do not continue if we can't read data
       fmt.Printf("FATAL: Unable to open src file at %s: %v\n", fullPath, err)
       cancel() // Stop the app
       return
   }
   defer inputFile.Close()

   scanner := bufio.NewScanner(inputFile)

   // ... [Rest of setup code] ...

   go func() {
      lineNum := 0
      for scanner.Scan() {
         // ... [Sending logic] ...
         lineNum++
      }

      // CRITICAL FIX: Check if scanner crashed
      if err := scanner.Err(); err != nil {
          fmt.Printf("Scanner Error: %v\n", err)
```

```
             } else {
                 fmt.Printf("Finished reading input file. Total lines: %d\n", lineNum)
             }

             // Send End-Of-Stream
             readingChan <- struct {
                line    string
                lineNum int
             }{line: "", lineNum: -1}

             r.Lock.Lock()
             r.DoneReading = true
             r.Lock.Unlock()
          }()

          // ... [Rest of the loop] ...
```

## Side Note: The "Huge PID" Bug

In your logs, you see PIDs like `824634747032`. This is because in `ReceiveTaskCompletion`, you are printing the **memory address** of the `reply` pointer, not the PID.

**Incorrect Line:**

Go

```go
app.LogFileChan <- fmt.Sprintf("... PID: %d ...", ..., reply, ...)
```

**Correct Line:**

Go

```go
// Use the PID stored in your TaskInformation map
app.LogFileChan <- fmt.Sprintf("... PID: %d ...", ..., app.TaskInformation[args.Stage][args
```

**User**

but this worked when auto scaling was off

**Gemini**

This behavior (system starting and immediately completing) almost always happens because the **Input Reader failed to open the file**, but your code allowed it to continue anyway.

When `os.Open` fails, `scanner.Scan()` returns false immediately, sending the "End of Stream" signal ( `-1` ) to the system. This causes all workers to shut down instantly.

### Why does it only happen with AutoScaling on?

It is likely due to your **command-line argument parsing**. Your parsing logic relies on counting from the end of the string ( `len(splits)-7` ). If you added an extra space in your command, or if the `AutoScale` flag shifted the argument count (e.g., `true` vs `false` shouldn't matter, but an accidental extra argument or space would), the parser will grab the wrong word for `HydfsSrcDirectory` .

If it tries to open a file that doesn't exist (or an empty string), it fails, prints an error to **Standard Out** (which you might have missed if you are looking at the log file), and then proceeds to shut down.

### The Fix

You need to make the application **Crash/Exit** if the input file cannot be found, rather than silently continuing.

Update the **Input Reader** section in `main.go` (around line 304 in your latest paste) to this:

Go

```go
        // --- INPUT READER (PRODUCER 2) ---
        wgProducers.Add(1)
        go func() {
            defer wgProducers.Done()

            // 1. Construct path and OPEN THE FILE
            fullPath := filepath.Join(dataDir, r.HydfsSrcDirectory)
            inputFile, err := os.Open(fullPath)

            // 2. CRITICAL FIX: Stop immediately if file is missing
            if err != nil {
                fmt.Printf("FATAL ERROR: Could not open source file: %s\nError: %v\n", fullPat
                // Force the app to stop waiting and exit
                cancel()
                // Send a signal to the completion channel so main loop unblocks if needed (op
                appCompletedChan <- true
                return
            }
            defer inputFile.Close()

            scanner := bufio.NewScanner(inputFile)
        // ... rest of your logic ...
```

### Verification

Check your **terminal console** (not the log file). You will likely see this message printed right at the start: `Unable to open src directory: ...`

If you see that, check the command string you are typing to start the app. Ensure there are no trailing spaces and that the number of arguments matches exactly what your switch statement expects.

**User**

what is wrong with this because it is still failing:

```
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 // [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 }
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.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) // 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)
localOutputFile.WriteString(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()
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 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() }()
// Tracks a tuple we failed to send and need to retry var pendingTuple *struct { line string lineNum int }
for { // Definition of the tuple variable for this iteration var tuple struct { line string lineNum int }
// --- SELECTION LOGIC --- if pendingTuple != nil { // 1. Prioritize the retry tuple = *pendingTuple pendingTuple = nil } else { // 2. Otherwise read from channel select { case <-ctx.Done(): for _, c := range tupleClients { c.Conn.Close() } return case t := <-readingChan: tuple = t // FIX 1: use '=' not ':=' } }
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 2: Handle empty task list without deadlocking channel if len(firstTaskList) == 0 { time.Sleep(50 * time.Millisecond) pendingTuple = &tuple // Save state continue // Retry loop }
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()) // FIX 3: Save tuple before continuing on dial fail pendingTuple = &tuple 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())
// Retry logic (already correct in your snippet, kept for clarity) pendingTuple = &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() inputFile.Close() 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 { // 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() } else if args.Rate > app.HighestRate { // 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() } } 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": //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 vm := splits[1] pid, _ := strconv.Atoi(splits[2])
// We need to release the lock before calling ReceiveFailure to avoid Deadlock, // because ReceiveFailure
acquires the lock itself. var foundTask bool var targetTask Task
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { for taskNum, info := range stage {
if vm == info.Ip.String() && pid == info.Pid { targetTask = Task{ TaskNumber: taskNum, Stage: stageNum, }
foundTask = true break } } if foundTask { break } } curApp.Lock.RUnlock()
if foundTask { // 1. Kill the physical process on the worker rpcWorkersLock.RLock() worker, ok :=
rpcWorkers[vm] // Use the vm string to find the client rpcWorkersLock.RUnlock()
if ok { var reply int err := worker.Call("Worker.KillTask", TaskID{ Task: targetTask.TaskNumber, Stage:
targetTask.Stage, }, &reply)
if err != nil { fmt.Println("Error killing task:", err) } else { fmt.Println("Task Killed via RPC.") } }
// 2. IMPORTANT: Tell the Leader to recover/reschedule immediately fmt.Println("Triggering Leader
Recovery...") var reply int // We call ReceiveFailure locally to update the map and send new IPs to everyone
curApp.ReceiveFailure(targetTask, &reply) } else { fmt.Println("Task not found to kill.") } 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**

You are still experiencing the "immediate shutdown" issue because **you did not apply the critical fix to the file opening logic.**

In your current code (lines 198-201), when `os.Open` fails, you print an error message **but let the code continue execution**.

Go

```go
// YOUR CURRENT CODE
inputFile, err := os.Open(filepath.Join(dataDir, r.HydfsSrcDirectory))
if err != nil {
   fmt.Println("Unable to open src directory: " + err.Error())
   // <--- THE BUG IS HERE: You don't stop! You keep going!
}
```

Because you continue, `inputFile` is invalid (or nil). The `bufio.NewScanner` creates a scanner that fails immediately. The scanner loop finishes instantly, sends the "End of Stream" signal, and your leader shuts down the whole application thinking the job is done.

### The Fix

You must **Force Stop** the setup if the file is missing.

**Replace lines 198-201 with this:**

Go

```go
    fullPath := filepath.Join(dataDir, r.HydfsSrcDirectory)
    inputFile, err := os.Open(fullPath)
    if err != nil {
        fmt.Printf("FATAL ERROR: Unable to open src file: %s\nError: %v\n", fullPath, err)
        // Stop the context so listeners close
        cancel()
        // Clean up previous steps
        _ = hydfsClient.Close()
        localOutputFile.Close()
        // Restart the loop or exit main
        continue
    }
```

### Secondary Issue: Why is the file failing to open?

If you are seeing the "Unable to open src directory" error, it is likely because your **Command Line Parser** is fragile.

Your parser uses `strings.Split(stdin, " ")`. If you accidentally type **two spaces** between arguments, or a trailing space at the end, `strings.Split` creates empty strings `""` in the array. Since you parse arguments from the *end* of the array (`len(splits)-7`), these empty strings shift your index, causing the program to look for the filename in the wrong place.

**Fix your `processStdin` function (Line 605) to filter out empty spaces:**

Go

```go
        stdin = strings.TrimSpace(stdin)
        // FIX: Handle multiple spaces correctly using Fields
        splits := strings.Fields(stdin) // .Fields handles multiple spaces automatically

        switch splits[0] {
        // ... rest of logic
```

### Visualizing the Failure Flow

**Summary of what to do:**

1. **Change `strings.Split` to `strings.Fields`** in `processStdin` to fix the argument parsing.

2. **Add `continue` or `return`** inside the `if err != nil` block when opening `inputFile` so the app doesn't try to run with a broken file handle.

**User**

but that print was never printed meaning it is probably something else

**Gemini**

You are facing a "Silent Failure" in the `bufio.Scanner`.

If the "Unable to open..." error never printed, that means `os.Open` **succeeded**. The file handle is valid. However, the application finishes instantly. This implies that `scanner.Scan()` returned `false` immediately on the very first attempt.

This usually happens for one of three reasons:

1. **The file is empty.**
2. **The path points to a Directory**, not a file (On Linux, `os.Open` on a directory succeeds, but `Scanner` fails immediately).
3. **Permissions issues** preventing reading.

Since you are not checking `scanner.Err()`, the scanner fails silently, your code sends the "End of Stream" signal (`-1`), and the leader shuts everything down gracefully.

## The Solution

You need to add debugging to see **what** file you are opening and **why** the scanner stopped.

Update your **Input Reader** logic (inside `main.go`) with this exact block. It adds the necessary error checks:

Go

```go
		// --- INPUT READER (PRODUCER 2) ---
		wgProducers.Add(1)
		go func() {
			defer wgProducers.Done()

			// DEBUG 1: Print exactly what we are trying to open
			fullPath := filepath.Join(dataDir, r.HydfsSrcDirectory)
			fmt.Printf("DEBUG: Attempting to open: '%s'\n", fullPath)

			inputFile, err := os.Open(fullPath)
			if err != nil {
				fmt.Printf("FATAL ERROR: Could not open source file: %v\n", err)
				cancel()
				appCompletedChan <- true
				return
			}
			defer inputFile.Close()

			// Check if it's actually a directory
			info, _ := inputFile.Stat()
			if info.IsDir() {
				fmt.Printf("FATAL ERROR: The path '%s' is a DIRECTORY, not a file!\n", fullPat
				cancel()
```

```
        appCompletedChan <- true
        return
    }

    scanner := bufio.NewScanner(inputFile)

    // ... [Lock/Task Setup logic remains same] ...

    go func() {
        lineNum := 0
        for scanner.Scan() {
            // ... [Your existing Loop Logic] ...

            lineNum++
        }

        // DEBUG 2: Check why the loop finished!
        if err := scanner.Err(); err != nil {
            fmt.Printf("FATAL SCANNER ERROR: %v\n", err)
        } else {
            fmt.Printf("Scanner finished. Total lines read: %d\n", lineNum)
        }

        // Send End-Of-Stream
        readingChan <- struct {
            line    string
            lineNum int
        }{line: "", lineNum: -1}

        r.Lock.Lock()
        r.DoneReading = true
        r.Lock.Unlock()
    }()

    // ... [Rest of Consumer Loop] ...
```

### Why does this happen only with AutoScaling?

If this works fine normally but fails when you add the `AutoScale` flag, your **command line parsing** is likely the culprit.

You are using `strings.Split(stdin, " ")`. If you typed the command like this (with extra spaces): `RainStorm ... true true 0.5 ...` (Note the double space)

`strings.Split` will create empty strings inside your slice. Because your parser logic counts from the end (`len(splits)-7`), the empty strings shift the index.

- Your `HydfsSrcDirectory` variable might be ending up as an empty string `""`.

- `filepath.Join(dataDir, "")` resolves to `.../data/`.
- `os.Open` on a directory **succeeds**.
- `scanner.Scan` on a directory **fails silently**.

**Fix the parsing in `processStdin` to be robust:** Replace:

Go

```go
splits := strings.Split(stdin, " ")
```

With:

Go

```go
splits := strings.Fields(stdin) // Automatically handles multiple spaces
```

**User**

ok now that is fixed, why am i getting this:
Received task completion for: 0, BUT should not have received this

**Gemini**

This error message confirms that your **Autoscaling (Downscaling)** logic is working, but you have a harmless "Race Condition" in your cleanup logic.

### The Cause: "I Forgot You Existed"

Here is the sequence of events causing this error:

1. **Leader (Downscaling):** You call `removeTask(stage, task)`.
2. **Leader:** You immediately **delete** the task from your `TaskInformation` map (Line 538). The Leader now has no memory of this task.
3. **Leader:** You send the `Worker.AutoscaleDown` RPC to the worker.
4. **Worker:** The worker shuts down the process.
5. **Worker (Being Polite):** Before dying, the worker sends a `ReceiveTaskCompletion` RPC back to the Leader to say "I'm finished!"
6. **Leader:** Receives the message. It looks for the task in `TaskInformation`, but **it's already gone** (deleted in Step 2).
7. **Leader:** Prints: `Received task completion for: 0, BUT should not have received this`.

### Is this a problem?

- **For Autoscaling: No.** It is a benign side effect. The task is gone, which is exactly what you wanted.
- **For `kill_task`: Yes, potentially.** If you kill a task to restart it, you must ensure the "Completion" message from the *dying* worker doesn't accidentally cancel the *new* replacement worker.

### The Fix

You should update `ReceiveTaskCompletion` to simply ignore completions for tasks that don't exist, rather than treating them as impossible errors.

Update your `ReceiveTaskCompletion` function in `main.go`:

Go

```go
func (app *RainStorm) ReceiveTaskCompletion(args TaskID, reply *int) error {
    app.Lock.Lock()
    defer app.Lock.Unlock()

    // Check if task exists in our map
    if info, exists := app.TaskInformation[args.Stage][args.Task]; exists {

       // --- [CRITICAL SAFETY CHECK] ---
       // Only process completion if it comes from the CURRENT worker.
       // If an old, killed worker reports completion, we must ignore it
       // so we don't kill the new replacement task!
       // (Assuming your TaskID args or the context allows checking IP,
       //  otherwise, this relies on the fact that 'removeTask' deletes the key first).

       app.LogFileChan <- fmt.Sprintf("Task Completed TaskID: %d Stage: %d VM: %s PID: %d\n
           args.Task, args.Stage, info.Ip.String(), *reply)

       delete(app.TaskInformation[args.Stage], args.Task)
       app.sendIps()

       if len(app.TaskInformation[args.Stage]) == 0 {
          app.sendStageCompletion(args.Stage)
          if args.Stage+1 == app.NumStages {
             appCompletedChan <- true
          }
       }

    } else {
       // CASE: The task was already removed (e.g., via Autoscaling).
       // This is expected behavior. Just log it as info/debug.
       // Do NOT panic or print a scary error.
       fmt.Printf("Info: Received completion for removed Task %d (Stage %d). Ignoring.\n",
    }
    return nil
}
```

## Summary

You are seeing this because the Leader deletes the record slightly faster than the Worker can say goodbye. It proves your `removeTask` logic is actually executing! You can treat this log message as "Task successfully downscaled."

**User**

this was the logfile for the application:

```
2025-12-07 20:48:25: Started RainStorm Application 2025-12-07 20:48:25: Starting Task at VM:
172.22.94.231 PID: 22172 op_exe: Filter 2025-12-07 20:48:25: Starting Task at VM: 172.22.94.231 PID: 22176
op_exe: Filter 2025-12-07 20:48:25: Starting Task at VM: 172.22.94.231 PID: 22181 op_exe: Filter 2025-12-07
20:48:25: Starting Task at VM: 172.22.94.231 PID: 22185 op_exe: Transform 2025-12-07 20:48:25: Starting
Task at VM: 172.22.94.231 PID: 22190 op_exe: Transform 2025-12-07 20:48:25: Starting Task at VM:
172.22.94.231 PID: 22195 op_exe: Transform 2025-12-07 20:48:26: Rate: 32.99 TaskID: 0 Stage 0 2025-12-07
20:48:26: Downscaling Stage: 0 Rate: 32.99 2025-12-07 20:48:26: Task Completed TaskID: 0 Stage: 0 VM:
172.22.94.231 PID: 824634322800 op_exe: Filter 2025-12-07 20:48:27: Rate: 0.00 TaskID: 1 Stage 0 2025-
12-07 20:48:27: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:27: Rate: 0.00 TaskID: 2 Stage 1 2025-
12-07 20:48:27: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:27: Rate: 1.00 TaskID: 2 Stage 0 2025-12-
07 20:48:27: Rate: 0.00 TaskID: 0 Stage 1 2025-12-07 20:48:27: Downscaling Stage: 0 Rate: 1.00 2025-12-07
20:48:27: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:28: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:28: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:28: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:28: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:28: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:28: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:29: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:29: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:29: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:29: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:29: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:29: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:30: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:30: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:30: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:30: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:30: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:30: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:31: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:31: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:31: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:31: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:31: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:31: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:32: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:32: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:32: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:32: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:32: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:32: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:33: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:33: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:33: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:33: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:33: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:33: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:34: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:34: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:34: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:34: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:34: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:34: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:35: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:35: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:35: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:35: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:35: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:35: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:36: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:36: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:36: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:36: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:36: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:36: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:37: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:37: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:37: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:37: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:37: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:37: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:38: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:38: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:38: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:38: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:38: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:38: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:39: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:39: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:39: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:39: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:39: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:39: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:40: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:40: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:40: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
```

```
20:48:40: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:40: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:40: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:41: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:41: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07 20:48:41: Downscaling Stage: 1 Rate: 0.00 2025-12-07
20:48:41: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:41: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:41: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:42: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:42: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:42: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:42: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:42: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:42: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:43: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:43: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07 20:48:43: Downscaling Stage: 1 Rate: 0.00 2025-12-07
20:48:43: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:43: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:43: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:44: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:44: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:44: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:44: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:44: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:44: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:45: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:45: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07 20:48:45: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:45: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07 20:48:45: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:45: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:46: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:46: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:46: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:46: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:46: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:46: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:47: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:47: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:47: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:47: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:47: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:47: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:48: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:48: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07 20:48:48: Downscaling Stage: 1 Rate: 0.00 2025-12-07
20:48:48: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07 20:48:48: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:48: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:49: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:49: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:49: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:49: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:49: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:49: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:50: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:50: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:50: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:50: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:50: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:50: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:51: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:51: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07 20:48:51: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:51: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:51: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:51: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:52: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:52: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07 20:48:52: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:52: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:52: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:52: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:53: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:53: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07 20:48:53: Downscaling Stage: 0 Rate: 0.00 2025-12-07
20:48:53: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:53: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:53: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:54: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:54: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:54: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:54: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:54: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:54: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:55: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:55: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:55: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:55: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:55: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:55: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:56: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:56: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:56: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:56: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:56: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:56: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:57: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:57: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:57: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:57: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:57: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
```

```
20:48:57: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:58: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:58: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:58: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:58: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:58: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:58: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:59: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:48:59: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:48:59: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:48:59: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:48:59: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:48:59: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:00: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:00: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:00: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:00: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:00: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:00: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:01: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:01: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:01: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:01: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:01: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:01: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:02: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:02: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:02: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:02: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:02: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:02: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:03: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:03: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:03: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:03: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:03: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:03: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:04: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:04: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:04: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:04: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:04: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:04: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:05: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:05: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:05: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:05: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:05: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:05: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:06: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:06: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:06: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:06: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:06: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:06: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:07: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07
20:49:07: Downscaling Stage: 1 Rate: 0.00 2025-12-07 20:49:07: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:07: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:07: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:07: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:08: Rate: 0.00 TaskID: 2 Stage 0 2025-12-07
20:49:08: Rate: 0.00 TaskID: 2 Stage 1 2025-12-07 20:49:08: Downscaling Stage: 1 Rate: 0.00 2025-12-07
20:49:08: Downscaling Stage: 0 Rate: 0.00 2025-12-07 20:49:08: Rate: 0.00 TaskID: 1 Stage 0 2025-12-07
20:49:08: Downscaling Stage: 0 Rate: 0.00
```

**Gemini**

Based on the logs, your system is entering a **"Death Spiral"** caused by aggressive autoscaling thresholds.

## The Problem: "Infanticide" and Bad Thresholds

Your logs reveal two critical configuration issues that are causing your system to kill tasks as soon as they are born.

### 1. Your `LowestRate` Threshold is too High

Look at this specific line in your log: `2025-12-07 20:48:26: Downscaling Stage: 0 Rate: 32.99`

You are downscaling (killing a task) when it is processing **33 items per second**. This implies you launched the application with a `LowestRate` argument that is **higher than 33** (e.g., maybe 100?). Because 32.99 is lower than your threshold, the Leader thinks the task is "underutilized" and kills it.

### 2. The "Zero-Rate" Death Spiral

Immediately after that, the logs show: `2025-12-07 20:48:27: Downscaling Stage: 1 Rate: 0.00`

When a new task starts, it has processed **0 items**, so its rate is naturally **0.00**.

1. Task Starts (Rate 0.00).
2. Leader sees `0.00 < LowestRate`.
3. Leader kills the task immediately.
4. Task restarts -> Rate 0.00 -> Killed again.

### The Fix

You need to prevent the Leader from killing tasks that are just "warming up."

Modify your `ReceiveRateUpdate` function to include a **Grace Period** or a **Zero-Check**.

**Update `ReceiveRateUpdate` in `main.go`:**

Go

```go
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
    app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n", args.Rate, args.Task

    if app.AutoScale {
       // FIX 1: Ignore 0.0 rate updates to prevent killing new tasks instantly.
       // (Real idle tasks will eventually report a tiny non-zero number or you can handle
       // strict 0.0 with a timestamp check, but ignoring 0.0 is the safest quick fix).
       if args.Rate <= 0.01 {
           return nil
       }

       // CASE 1: Rate is LOW -> DOWNSCALE
       if args.Rate < app.LowestRate {
          app.Lock.Lock()
          // Safety: Never kill the last worker
          if len(app.TaskInformation[args.Stage]) > 1 {
             app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.Stag
             app.removeTask(args.Stage)
          }
          app.Lock.Unlock()

       // CASE 2: Rate is HIGH -> UPSCALE
       } else if args.Rate > app.HighestRate {
          app.Lock.Lock()
          taskNum := app.NextTaskNum[args.Stage]
          app.NextTaskNum[args.Stage]++
          app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage, ar
          app.addTask(args.Stage, taskNum)
          app.sendIps()
          app.Lock.Unlock()
       }
    }
    return nil
}
```

## Next Step for You

1. **Apply the code fix** above (ignoring `Rate <= 0.01`).
2. **Check your command line arguments.** Ensure you are setting reasonable thresholds.
    - Set `LowestRate` to something small like **0.5** or **1.0**.
    - Set `HighestRate` to something reachable like **1000.0**.
    - *Example Command:* `RainStorm 2 2 Filter ... 0.5 1000.0`

**User**

how can i make it so that i only auto scale every 3 seconds and wait for system to settle a little bit

**Gemini**

To implement a "cooldown" or "settling" period, you need to track the last time an autoscaling action occurred for each stage.

Here is how to modify your code to enforce a 3-second wait between scaling events.

### Step 1: Update the Struct

Add a `LastAutoscaleTime` map to your `RainStorm` struct definition.

Go

```go
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

    // NEW FIELD
    LastAutoscaleTime        map[int]time.Time
}
```

### Step 2: Initialize the Map

In `main()`, right before you start the initial tasks (around line 180 in your previous code), initialize this map. Setting it to `time.Now()` ensures the system waits 3 seconds after startup before doing anything.

Go

```
       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)

       // --- NEW INITIALIZATION ---
       r.LastAutoscaleTime = make(map[int]time.Time)
       for i := range r.NumStages {
           r.LastAutoscaleTime[i] = time.Now() // Wait 3s from start before autoscaling
       }
       // --------------------------

       r.DoneReading = false
       r.initWorker()
```

### Step 3: Update `ReceiveRateUpdate`

Modify the function to check the time before acting. If 3 seconds haven't passed since the last action for that stage, we simply ignore the request.

Go

```go
func (app *RainStorm) ReceiveRateUpdate(args RmUpdate, reply *int) error {
    app.LogFileChan <- fmt.Sprintf("Rate: %.2f TaskID: %d Stage %d\n", args.Rate, args.Task

    if app.AutoScale {
       // Ignore zero rates to prevent "Infanticide" of new tasks
       if args.Rate <= 0.01 {
           return nil
       }

       // DOWNSCALE
       if args.Rate < app.LowestRate {
          app.Lock.Lock()
          // Check Cooldown: Has it been 3 seconds?
          if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second {
              if len(app.TaskInformation[args.Stage]) > 1 {
                 app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Rate: %.2f\n", args.
                 app.removeTask(args.Stage)
                 // Reset the timer
                 app.LastAutoscaleTime[args.Stage] = time.Now()
              }
          }
          app.Lock.Unlock()

       // UPSCALE
```

```
        } else if args.Rate > app.HighestRate {
            app.Lock.Lock()
            // Check Cooldown: Has it been 3 seconds?
            if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second {
                taskNum := app.NextTaskNum[args.Stage]
                app.NextTaskNum[args.Stage]++
                app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage
                app.addTask(args.Stage, taskNum)
                app.sendIps()
                // Reset the timer
                app.LastAutoscaleTime[args.Stage] = time.Now()
            }
            app.Lock.Unlock()
        }
    }
    return nil
}
```

## Why this helps

1. **System Settling:** When you add a task, the average rate drops temporarily. Without a delay, the system might panic and try to add *another* task immediately. The 3-second pause lets the new worker come online and the rate stabilize.
2. **Prevents Flapping:** It stops the system from adding a task (upscaling) and immediately deleting it (downscaling) if the rate is oscillating near the threshold.

**User**

the worker is facing a similar issue to the leader from before where it seems like the channel is getting full when failing to send. i think this is causing it to block. check this and apply a similar fix as above

**Gemini**

Based on your description, the `Worker`'s output channel (`taskOutputs`) is likely filling up because failed tuple sends are being pushed back into the channel using ephemeral goroutines (`go func(t) { ... worker.taskOutputs <- t }(out)`). When the system is under load or facing failures (like a disconnected peer), these retries fill the buffer, blocking the `tasks` from writing to `taskOutputs`, effectively halting the entire worker.

To fix this, we replace the "push-back to channel" logic with a local `pendingTuple` variable. This holds the failed tuple in memory and prioritizes retrying it in the next loop iteration, preventing the channel from overflowing.

Here is the updated `main.go` for the Worker:

Go

```go
package main

import (
        "bufio"
        "bytes"
        "context"
        "encoding/csv"
        "errors"
        "fmt"
        . "g14-mp4/RainStorm/resources"
        "g14-mp4/mp3/resources"
        "io"
        "net"
        "net/rpc"
        "os"
        "os/exec"
        "path/filepath"
        "slices"
        "strconv"
        "strings"
        "sync"
        "time"
)

type localTask struct {
        cmd            *exec.Cmd
        input          io.WriteCloser // To send tuples to tasks (receives data from tcp)
        output         io.ReadCloser  // To send tuples to the next stage (sends data throu
        inputRate      int
```

```
	startTime      time.Time
	lastCheckTime  time.Time
	lastInputCount int
	logFile        *os.File
}

type taskOutput struct {
	tupleId int
	taskId  TaskID
	output  string
}

type logRequest struct {
	fileName string
	data     string
}

type Worker struct {
	rainStormLeader    *rpc.Client // used to send task completions
	rainStormStartTime string
	hydfsClient        *rpc.Client
	hydfsDestFile      string
	lowWatermark       float64
	highWatermark      float64

	done        chan bool
	tasksLocker sync.RWMutex
	tasks       map[TaskID]*localTask

	ips           []map[int]*TaskInfo // ips of machines with [stage][task] indexing
	taskIDLocker  sync.RWMutex
	sortedTaskIDs [][]int // used to find the task # within a given stage
	logChan       chan logRequest

	taskOutputs     chan taskOutput
	connections     map[string]*WorkerClient
	connectionsLock sync.RWMutex
	stageOperations []Operation

	tuplesLock     sync.Mutex
	receivedTuples map[string]bool // key = taskId-TupleId, value is dummy
	tupleSendConn  net.Conn

	finishedStages map[int]bool // Add this
}
```

```
const clientTimeout = time.Second * 3
const ACK = "ACK"

func main() {
        leader, err := rpc.Dial("tcp", "fa25-cs425-1401.cs.illinois.edu"+IntroducePort)
        if err != nil {
                fmt.Println(err)
                return
        }
        var reply int
        err = leader.Call("WorkerIps.AddWorker", getOutboundIP(), &reply)
        if err != nil {
                fmt.Println(err)
                return
        }
        _ = leader.Close()
        hydfsClient, err := rpc.Dial("tcp", "localhost:8011") // connect to our own HydFS c
        if err != nil {
                fmt.Println(err)
                return
        }
        for {
                server := rpc.NewServer()
                if err != nil {
                        time.Sleep(1 * time.Second)
                        continue // try again
                }
                worker := Worker{
                        hydfsClient:    hydfsClient,
                        done:           make(chan bool),
                        tasks:          make(map[TaskID]*localTask),
                        taskOutputs:    make(chan taskOutput, 100),
                        connections:    make(map[string]*WorkerClient),
                        receivedTuples: make(map[string]bool),
                        logChan:        make(chan logRequest, 1000),
                        finishedStages: make(map[int]bool),
                }

                ctx, cancel := context.WithCancel(context.Background())
                var wg sync.WaitGroup

                err = server.Register(&worker)
                if err != nil {
                        time.Sleep(1 * time.Second)
                        continue
                }
```

```
		leaderListener, err := net.Listen("tcp", AssignmentPort)
		if err != nil {
			//fmt.Println(err)
			time.Sleep(1 * time.Second)
			continue
		}
		go server.Accept(leaderListener)

		// Goroutine for sending out tuples
		wg.Add(1)
		go func() {
			defer wg.Done()
			var pendingTuple *taskOutput

			for {
				var out taskOutput

				// Check if we have a pending retry first
				if pendingTuple != nil {
					select {
					case <-ctx.Done():
						return
					default:
					}
					out = *pendingTuple
					pendingTuple = nil
				} else {
					// Otherwise read from channel
					select {
					case <-ctx.Done():
						return
					case val := <-worker.taskOutputs:
						out = val
					}
				}

				nextStage := out.taskId.Stage + 1
				// Remote log
				worker.logChan <- logRequest{
					fileName: fmt.Sprintf("%s_%d-%d", worker.rainStormS
					data:     fmt.Sprintf("PROCESSED,%s-%d,%s\n", out.t
				}

				// local log
				worker.tasksLocker.RLock()
				if t, ok := worker.tasks[out.taskId]; ok && t.logFile != ni
```

```
					_, _ = fmt.Fprintln(t.logFile, "OUTPUT: ", out.outp
				}
				worker.tasksLocker.RUnlock()
				if nextStage < len(worker.ips) { // send it to the next sta
					key := out.output
					if worker.stageOperations[nextStage].Name == Aggreg
						hashIndex, err := strconv.Atoi(worker.stage
						if err != nil {
							hashIndex = 0
						}
						reader := csv.NewReader(strings.NewReader(o
						tuple, err := reader.Read()
						if err == nil && hashIndex < len(tuple) {
							key = tuple[hashIndex]
						}
					}

					// Find which client gets the next tuple
					worker.taskIDLocker.RLock()
					nextStageTasks := worker.sortedTaskIDs[nextStage]
					hash := HashString(key)
					if hash < 0 { // make sure hash is positive
						hash = -hash
					}
					nextTask := nextStageTasks[hash%len(nextStageTasks)

					nextWorker := worker.ips[nextStage][nextTask].Ip.St
					worker.taskIDLocker.RUnlock()

					worker.connectionsLock.RLock()
					client, ok := worker.connections[nextWorker]
					worker.connectionsLock.RUnlock()
					if !ok { // new connection,
						// try connecting
						conn, err := net.Dial("tcp", nextWorker+Tup
						if err != nil {
							time.Sleep(100 * time.Millisecond)
							pendingTuple = &out
							continue
						}
						newClient := &WorkerClient{
							Conn: conn,
							Buf:  bufio.NewReader(conn),
						}

						worker.connectionsLock.Lock()
						// Make sure the client wasn't already adde
						if existing, exists := worker.connections[n
							// Already exists, just use that on
							client = existing
							conn.Close()
						} else {
							client = newClient
							worker.connections[nextWorker] = cl
						}
						worker.connectionsLock.Unlock()
					}

					// Send the tuple
					_ = client.Conn.SetWriteDeadline(time.Now().Add(cli
					// Id-Id, stage, task, data
					_, err = fmt.Fprintf(client.Conn, "%s-%d,%d,%d,%s\n

					if err != nil { // Write didn't go through, disconn
						_ = client.Conn.Close()
						worker.connectionsLock.Lock()
						delete(worker.connections, nextWorker)
						worker.connectionsLock.Unlock()
						time.Sleep(100 * time.Millisecond) // Wait
						pendingTuple = &out                // Save
						continue
					}

					// Wait for the ack
					_ = client.Conn.SetReadDeadline(time.Now().Add(clie
					ack, err := client.Buf.ReadString('\n')
					expectedAck := fmt.Sprintf("%s-%d-%s", out.taskId.S
					if err != nil || strings.TrimSpace(ack) != expected
						time.Sleep(100 * time.Millisecond) // Wait
						pendingTuple = &out                // didn'
						continue
					}
				} else { // output data to the distributed file system
					// Send the tuple to the leader, they will write to
					if worker.tupleSendConn == nil {
						fmt.Println("CRITICAL ERROR: tupleSendConn
						time.Sleep(100 * time.Millisecond) // Wait
						pendingTuple = &out                // Save
						time.Sleep(100 * time.Millisecond) // Wait
						continue
					}
					_, err = fmt.Fprintln(worker.tupleSendConn, out.out
			if err != nil {
				fmt.Println("error sending tuple to leader"
				time.Sleep(100 * time.Millisecond) // Wait
				pendingTuple = &out                // Save
				continue
			}
		}

	}
}()

// Goroutine for reading in tuples
tupleListener, err := net.Listen("tcp", TuplePort)
if err != nil {
	return
}
go func(listener net.Listener) {
	for {
		conn, err := listener.Accept()
		if err != nil {
			return
		}
		go func(conn net.Conn) {
			defer conn.Close()
			reader := bufio.NewReader(conn)
			for {
				tuple, err := reader.ReadString('\n')
				if err != nil {
					return // connection closed/failed
				}
				split := strings.SplitN(tuple, ",", 4)

				// find the correct task
				stage, err := strconv.Atoi(split[1])
				if err != nil {
					continue
				}
				task, err := strconv.Atoi(split[2])
				if err != nil {
					continue
				}
				targetTask := TaskID{Stage: stage, Task: ta

				// De-duplication
				worker.tuplesLock.Lock()
				if _, ok := worker.receivedTuples[split[0]]
```

```
							// Log rejection
							worker.tasksLocker.RLock()
							if t, ok := worker.tasks[targetTask
								_, _ = fmt.Fprintln(t.logFi
							}
							worker.tasksLocker.RUnlock()

							// We have already received this tu
							ackMsg := fmt.Sprintf("%s-%s\n", sp
							_, _ = fmt.Fprintf(conn, ackMsg)
							continue
						} else {
							worker.receivedTuples[split[0]] = t
						}
						worker.tuplesLock.Unlock()

						// write to task
						worker.tasksLocker.Lock()
						var taskInput io.Writer
						t, ok := worker.tasks[targetTask]
						if ok {
							taskInput = t.input
						}
						worker.tasksLocker.Unlock() // <--- RELEASE

						// 2. Perform Blocking I/O OUTSIDE the lock
						if taskInput != nil {
							_, err = io.WriteString(taskInput,
							if err != nil {
								continue // Write failed
							}

							// 3. Re-acquire Lock to update sta
							worker.tasksLocker.Lock()
							if t, ok := worker.tasks[targetTask
								if t.inputRate == 0 {
									t.startTime = time.
								}
								t.inputRate++
							}
							worker.tasksLocker.Unlock()
						} else {
							// Task didn't exist or input was n
							// Send ACK anyway to prevent sende
							// Or just continue (sender will re
							continue
						}
						if err != nil {
							continue // we weren't able to writ
						}
						// send the ack
						_ = conn.SetWriteDeadline(time.Now().Add(5
						ackMsg := fmt.Sprintf("%s-%s\n", split[0],
						_, err = fmt.Fprintf(conn, ackMsg)
						if err != nil {
							continue
						}
						//var r []resources.AppendReply
						//worker.hydfsClient.Go("Client.RemoteAppen
						//	RemoteName: fmt.Sprintf("%s_%d-%d",
						//	Content:    []byte(fmt.Sprintf("REC
						//}, &r, nil)
						worker.logChan <- logRequest{
							fileName: fmt.Sprintf("%s_%d-%d", w
							data:     fmt.Sprintf("RECEIVED,%s,
						}
					}
				}(conn)
			}
		}(tupleListener)

		// Local Resource Manager
		wg.Add(1)
		go func() {
			defer wg.Done()
			ticker := time.NewTicker(time.Second)
			defer ticker.Stop()

			for {
				select {
				case <-ctx.Done():
					return
				case <-ticker.C:
					worker.tasksLocker.RLock()
					for t, task := range worker.tasks {

						if task.startTime.IsZero() {
							continue
						}

						// If this is the first check, snapshot the
						if task.lastCheckTime.IsZero() {
```

```
                                        task.lastCheckTime = time.Now()
                                        task.lastInputCount = task.inputRat
                                        continue
                                }

                                // (Current Total - Old Total) / (Now - Old
                                now := time.Now()
                                duration := now.Sub(task.lastCheckTime).Sec

                                if duration > 0 {
                                        // Calculate tuples per second over
                                        tuplesReceivedSinceLastTick := floa
                                        rate := tuplesReceivedSinceLastTick

                                        // Update snapshot for the next tic
                                        task.lastCheckTime = now
                                        task.lastInputCount = task.inputRat

                                        // Send rate update to leader
                                        var r int
                                        worker.rainStormLeader.Go("RainStor
                                                Stage: t.Stage,
                                                Rate:  rate,
                                                Task:  t.Task,
                                        }, &r, nil)
                                }
                        }
                        worker.tasksLocker.RUnlock()
                }
        }
}()

// Goroutine for writing to log files
wg.Add(1)
go func() {
        defer wg.Done()
        // Map to buffer data for each file: map[filename]*bytes.Buffer
        buffers := make(map[string]*bytes.Buffer)
        ticker := time.NewTicker(500 * time.Millisecond) // Flush interval
        defer ticker.Stop()
        for {
                select {
                case <-ctx.Done():
                        return
                case <-ticker.C:
                        for name, buf := range buffers {
```

```
						if buf.Len() > 0 {
							var r []resources.AppendReply
							// Copy data to avoid race conditio
							data := make([]byte, buf.Len())
							copy(data, buf.Bytes())

							// Async RPC to append the batch
							worker.hydfsClient.Go("Client.Remot
								RemoteName: name,
								Content:    data,
							}, &r, nil)

							// Clear the buffer
							buf.Reset()
						}
					}
				case req := <-worker.logChan:
					// Get or create a buffer for this specific file
					buf, ok := buffers[req.fileName]
					if !ok {
						buf = new(bytes.Buffer)
						buffers[req.fileName] = buf
					}
					// Add the log line to the buffer
					buf.WriteString(req.data)
					// Flush when the buffer gets too big
					if buf.Len() > 10240 {
						var r []resources.AppendReply
						data := make([]byte, buf.Len())
						copy(data, buf.Bytes())
						worker.hydfsClient.Go("Client.RemoteAppend"
							RemoteName: req.fileName,
							Content:    data,
						}, &r, nil)
						buf.Reset()
					}
				}
			}
		}()
		<-worker.done
		println("This job finished")
		cancel()
		_ = leaderListener.Close()
		_ = worker.rainStormLeader.Close()
		_ = worker.tupleSendConn.Close()
		_ = tupleListener.Close()
```

```
                wg.Wait()
                time.Sleep(1 * time.Second) // wait for os to release port 8021
        }

}

// getOutboundIP gets the preferred outbound Ip of this machine, source: https://stackoverf
func getOutboundIP() net.IP {
        conn, err := net.Dial("udp", "8.8.8.8:80")
        if err != nil {
                panic(err)
        }
        defer func() {
                _ = conn.Close()
        }()

        localAddr := conn.LocalAddr().(*net.UDPAddr)
        return localAddr.IP.To4()
}

func (w *Worker) ReceiveFinishedStage(stage int, reply *int) error {
        w.tasksLocker.Lock()
        w.finishedStages[stage] = true // Record the completion
        var inputsToClose []io.WriteCloser
        for key, value := range w.tasks {
                if key.Stage == stage+1 {
                        inputsToClose = append(inputsToClose, value.input)
                }
        }
        w.tasksLocker.Unlock()
        println("Received stage ", stage, " completed")
        for _, input := range inputsToClose {
                _ = input.Close()
        }
        if stage == len(w.ips)-1 {
                w.done <- true
        }
        return nil
}

func (w *Worker) AutoscaleDown(t TaskID, reply *int) error {
        w.tasksLocker.RLock()
        defer w.tasksLocker.RUnlock()
        _ = w.tasks[t].input.Close()
        return nil
}
```

```
func (w *Worker) Initialize(args InitArgs, reply *int) error {
	w.stageOperations = args.Ops
	w.rainStormStartTime = args.Time.Format("20060102150405")
	w.hydfsDestFile = args.HyDFSDestFile
	w.lowWatermark = args.LowWatermark
	w.highWatermark = args.HighWatermark
	rainStormLeader, _ := rpc.Dial("tcp", "fa25-cs425-1401.cs.illinois.edu"+GlobalRMPor
	w.rainStormLeader = rainStormLeader
	tupleSendConn, err := net.Dial("tcp", "fa25-cs425-1401.cs.illinois.edu"+TuplePort)
	if err != nil {
		fmt.Println(err)
	}
	w.tupleSendConn = tupleSendConn
	fmt.Println("Started Rainstorm task")
	return nil
}

func (w *Worker) ReceiveIPs(ips []map[int]*TaskInfo, reply *int) error {
	println("received new IPS")
	w.taskIDLocker.Lock()
	defer w.taskIDLocker.Unlock()

	w.ips = ips
	w.sortedTaskIDs = make([][]int, len(ips))

	// update sortedTaskIDs
	for stage, tasks := range ips {
		w.sortedTaskIDs[stage] = make([]int, 0, len(tasks))
		for task := range tasks {
			w.sortedTaskIDs[stage] = append(w.sortedTaskIDs[stage], task)
		}
		slices.Sort(w.sortedTaskIDs[stage])
	}
	return nil
}

func (w *Worker) AddTask(t Task, reply *int) error {
	// Set up the task and its pipes
	cmdArgs := strings.Fields(t.Executable.Args)
	task := exec.Command(string(t.Executable.Name), cmdArgs...)
	taskStdin, err := task.StdinPipe()
	if err != nil {
		return err
	}
```

```
        taskStdout, err := task.StdoutPipe()
        if err != nil {
                return err
        }

        err = task.Start()
        if err != nil {
                return err
        }
        homeDir, _ := os.UserHomeDir()
        logDir := filepath.Join(homeDir, "taskLogs")
        _ = os.MkdirAll(logDir, 0755) // Create dir if missing
        logFile, err := os.OpenFile(filepath.Join(logDir, fmt.Sprintf("task_%d_%d_%s", t.St
        if err != nil {
                return err
        }
        _, _ = fmt.Fprintln(logFile, "Starting task:", t.Executable.Name, "with pid: ", tas

        // Connect the task's pipe to the channel
        tId := taskToTaskId(t)
        go func(pipe io.Reader, t TaskID, c chan<- taskOutput, cmd *exec.Cmd) {
                scanner := bufio.NewScanner(pipe)
                counter := 0
                for scanner.Scan() {
                        c <- taskOutput{
                                tupleId: counter,
                                taskId:  t,
                                output:  scanner.Text(),
                        }
                        counter++
                }

                if scanner.Err() != nil {
                        fmt.Println("Scanner error:", scanner.Err())
                }

                err = cmd.Wait()
                if err == nil {
                        var reply int
                        println("told leader im done", t.String())
                        err = w.rainStormLeader.Call("RainStorm.ReceiveTaskCompletion", t,
                        if err != nil {
                                println(err.Error())
                        }
                        println("leader responded", t.String())
                } else {
```

```
			var reply int
			_ = w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &reply)
			fmt.Printf("Task %v failed: %v\n", t, err)
		}
		w.tasksLocker.Lock()
		if storedTask, ok := w.tasks[t]; ok && storedTask.cmd == task {
			_ = storedTask.logFile.Close()
			delete(w.tasks, t)
		}
		w.tasksLocker.Unlock()
	}(taskStdout, tId, w.taskOutputs, task)

	// Add the task to the map
	w.tasksLocker.Lock()
	w.tasks[tId] = &localTask{
		cmd:     task,
		input:   taskStdin,
		output:  taskStdout,
		logFile: logFile,
	}
	w.tasksLocker.Unlock()

	// Check if the task has any tuples it needs to recover

	// First, check if this is the first time this task is getting created
	var createReply []resources.AddFileReply
	taskLogFile := fmt.Sprintf("%s_%d-%d", w.rainStormStartTime, t.Stage, t.TaskNumber)
	err = w.hydfsClient.Call("Client.RemoteCreate", &resources.RemoteFileArgs{
		RemoteName: taskLogFile,
		Content:    make([]byte, 0),
	}, &createReply)
	if err != nil {
		return err
	}
	recoveredTask := false
	for _, fileReply := range createReply {
		var e *resources.FileAlreadyExistsError
		if errors.As(fileReply.Err, &e) { // file already exists, so this is a reco
			recoveredTask = true
		}
	}

	if recoveredTask {
		// Need to go through the log file and get all the tuples that haven't been
		var contents []byte
		err = w.hydfsClient.Call("Client.RemoteGet", taskLogFile, &contents)
```

```
			scanner := bufio.NewScanner(bytes.NewReader(contents))

			// Mark all processed tuples
			w.tuplesLock.Lock()
			tuples := make(map[string]string)      // Key=ID, Value=tuple
			tupleStatus := make(map[string]string) // Key=ID, Value="RECEIVED" or "PROC
			for scanner.Scan() {
				splits := strings.SplitN(scanner.Text(), ",", 3)
				if len(splits) != 3 {
					continue
				}
				id := splits[1]
				w.receivedTuples[id] = true
				if splits[0] == "PROCESSED" {
					tupleStatus[id] = "PROCESSED"
				} else if splits[0] == "RECEIVED" {
					if _, exists := tupleStatus[id]; !exists {
						tupleStatus[id] = "RECEIVED"
						tuples[id] = splits[2]
					}
				}
			}
			w.tuplesLock.Unlock()

			// Add all unmarked tuples
			for _, tuple := range tuples {
				_, err = io.WriteString(taskStdin, tuple+"\n")
				if err != nil {
					fmt.Println("Error writing tuple to task ", tuple, err)
				}
			}
		}

		w.tasksLocker.RLock()
		isPrevStageDone := w.finishedStages[t.Stage-1]
		w.tasksLocker.RUnlock()

		if isPrevStageDone {
			// If Stage 0 is done, we must close the input immediately
			// so the task processes what it has and then exits.
			_ = taskStdin.Close()
		}

		*reply = task.Process.Pid
		return nil
	}
```

```go
func (w *Worker) KillTask(t TaskID, reply *int) error {
	w.tasksLocker.Lock()
	defer w.tasksLocker.Unlock()
	task, ok := w.tasks[t]
	fmt.Println("Received kill task")
	if ok {
		fmt.Printf("Killing task: %v\n", task)
		_ = task.cmd.Process.Kill()
		_ = task.input.Close()
		_ = task.output.Close()
		delete(w.tasks, t)
	}
	return nil
}

func taskToTaskId(t Task) TaskID {
	return TaskID{
		Stage: t.Stage,
		Task:  t.TaskNumber,
	}
}
```

**User**

here is the new leader code. for some reason the the frirst 2 downscaling events reach the worker but the remaining dont and those are what cause the blocking:

```
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 // [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
LastAutoscaleTime map[int]time.Time }
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.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.LastAutoscaleTime = make(map[int]time.Time) for i := range r.NumStages { r.LastAutoscaleTime[i] =
time.Now() // Wait 3s from start before autoscaling }
```

```
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) // 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)
localOutputFile.WriteString(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()
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 lineNum int }, 100)
go func() { lineNum := 0 for scanner.Scan() { readingChan <- struct { line string lineNum int }{line:
scanner.Text(), lineNum: lineNum} lineNum++ } if err := scanner.Err(); err != nil { fmt.Printf("FATAL SCANNER
ERROR: %v\n", err) } else { fmt.Printf("Scanner finished. Total lines read: %d\n", lineNum) } readingChan <-
struct { line string lineNum int }{line: "", lineNum: -1} r.Lock.Lock() r.DoneReading = true r.Lock.Unlock() }()
// Tracks a tuple we failed to send and need to retry var pendingTuple *struct { line string lineNum int }
for { // Definition of the tuple variable for this iteration var tuple struct { line string lineNum int }
// --- SELECTION LOGIC --- if pendingTuple != nil { // 1. Prioritize the retry tuple = *pendingTuple pendingTuple
= nil } else { // 2. Otherwise read from channel select { case <-ctx.Done(): for _, c := range tupleClients {
c.Conn.Close() } return case t := <-readingChan: tuple = t // FIX 1: use '=' not ':=' } }
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 2: Handle empty task list without deadlocking channel if len(firstTaskList) == 0 { time.Sleep(50 *
time.Millisecond) pendingTuple = &tuple // Save state continue // Retry loop }
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()) // FIX 3: Save tuple before continuing on dial fail
pendingTuple = &tuple 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())
// Retry logic (already correct in your snippet, kept for clarity) pendingTuple = &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() inputFile.Close() 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 <= 0.01 { return nil } if args.Rate < app.LowestRate { // remove a task
from this stage app.Lock.Lock() if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second { if
len(app.TaskInformation[args.Stage]) > 1 { app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Task: %d
Rate: %.2f\n", args.Stage, args.Task, args.Rate) fmt.Printf("Downscaling Stage: %d Task: %d Rate: %.2f\n",
args.Stage, args.Task, args.Rate) app.removeTask(args.Stage) // Reset the timer
app.LastAutoscaleTime[args.Stage] = time.Now() } } app.Lock.Unlock() } else if args.Rate > app.HighestRate {
// add a task to this stage app.Lock.Lock() if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second
{ 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() // Reset the timer app.LastAutoscaleTime[args.Stage] = time.Now() } 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": //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 vm := splits[1] pid, _ := strconv.Atoi(splits[2])
// We need to release the lock before calling ReceiveFailure to avoid Deadlock, // because ReceiveFailure
acquires the lock itself. var foundTask bool var targetTask Task
curApp.Lock.RLock() for stageNum, stage := range curApp.TaskInformation { for taskNum, info := range stage {
if vm == info.Ip.String() && pid == info.Pid { targetTask = Task{ TaskNumber: taskNum, Stage: stageNum, }
foundTask = true break } } if foundTask { break } } curApp.Lock.RUnlock()
if foundTask { // 1. Kill the physical process on the worker rpcWorkersLock.RLock() worker, ok :=
rpcWorkers[vm] // Use the vm string to find the client rpcWorkersLock.RUnlock()
if ok { var reply int err := worker.Call("Worker.KillTask", TaskID{ Task: targetTask.TaskNumber, Stage:
targetTask.Stage, }, &reply)
if err != nil { fmt.Println("Error killing task:", err) } else { fmt.Println("Task Killed via RPC.") } }
// 2. IMPORTANT: Tell the Leader to recover/reschedule immediately fmt.Println("Triggering Leader
Recovery...") var reply int // We call ReceiveFailure locally to update the map and send new IPs to everyone
curApp.ReceiveFailure(targetTask, &reply) } else { fmt.Println("Task not found to kill.") } 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**

The reason your system blocks after the first few downscaling events is that **RPC calls are synchronous and holding the Global Lock.**

When your Leader decides to remove a task, it holds the main `app.Lock`. Inside this lock, it does two dangerous things:

1. **`app.sendIps()`**: It waits for **every single worker** to acknowledge the new routing table. If just one worker is slow (processing a heavy tuple batch) or stuck, the Leader freezes.

2. `Worker.AutoscaleDown`: It calls the worker to kill the task and **waits for a reply**. If the worker is busy cleaning up, the Leader freezes.

Because the Leader holds `app.Lock` during this freeze, it cannot process any other messages (Rate Updates, Task Completions, etc.), causing the entire system to deadlock.

### The Fix

1. **Make `AutoscaleDown` Asynchronous:** The Leader has already deleted the task from its own map. It doesn't need to wait for the worker to confirm the kill. Use `Go` (Fire-and-Forget) instead of `Call`.
2. **Add a Timeout to `sendIps`:** Do not wait forever for workers to update their routing tables. If a worker is slow, log a warning and move on so the Leader stays responsive.

Here is the corrected code:

Go

```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 // [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
    LastAutoscaleTime        map[int]time.Time
}

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("
         _ = 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()
```

```go
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.LastAutoscaleTime = make(map[int]time.Time)
for i := range r.NumStages {
   r.LastAutoscaleTime[i] = time.Now() // Wait 3s from start before autoscaling
}

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.Hydfs

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)
                        localOutputFile.WriteString(line)
                        outputChan <- line // Send. We know writer is alive until we close ch
                     }

                     if err != nil {
                        return
                     }
                  }
               }(conn)
            }
         }()

         // --- INPUT READER (PRODUCER 2) ---
         wgProducers.Add(1)
         go func() {
            defer wgProducers.Done()

            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
```

```
        lineNum int
    }, 100)

    go func() {
        lineNum := 0
        for scanner.Scan() {
            readingChan <- struct {
                line    string
                lineNum int
            }{line: scanner.Text(), lineNum: lineNum}
            lineNum++
        }
        if err := scanner.Err(); err != nil {
            fmt.Printf("FATAL SCANNER ERROR: %v\n", err)
        } else {
            fmt.Printf("Scanner finished. Total lines read: %d\n", lineNum)
        }
        readingChan <- struct {
            line    string
            lineNum int
        }{line: "", lineNum: -1}
        r.Lock.Lock()
        r.DoneReading = true
        r.Lock.Unlock()
    }()

    // Tracks a tuple we failed to send and need to retry
    var pendingTuple *struct {
        line    string
        lineNum int
    }

    for {
        // Definition of the tuple variable for this iteration
        var tuple struct {
            line    string
            lineNum int
        }

        // --- SELECTION LOGIC ---
        if pendingTuple != nil {
            // 1. Prioritize the retry
            tuple = *pendingTuple
            pendingTuple = nil
        } else {
            // 2. Otherwise read from channel
```

```
   select {
   case <-ctx.Done():
      for _, c := range tupleClients {
         c.Conn.Close()
      }
      return
   case t := <-readingChan:
      tuple = t // FIX 1: use '=' not ':='
   }
}

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 2: Handle empty task list without deadlocking channel
if len(firstTaskList) == 0 {
   time.Sleep(50 * time.Millisecond)
   pendingTuple = &tuple // Save state
   continue              // Retry loop
}

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())
```

```
                     // FIX 3: Save tuple before continuing on dial fail
                     pendingTuple = &tuple
                     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,

               _ = 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())

                  // Retry logic (already correct in your snippet, kept for clarity)
                  pendingTuple = &tuple
                  continue
               }

               numProcessed++
               expectedDuration := time.Duration((numProcessed / r.InputRate) * float64(time.
               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()
        inputFile.Close()
        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.Stag
    } else {
        workers.l.RLock()
        app.TaskInformation[task.Stage][task.TaskNumber].Ip = workers.ips[app.NextAvailableV
        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
        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
    if app.AutoScale {
        if args.Rate <= 0.01 {
            return nil
        }
        if args.Rate < app.LowestRate {
            // remove a task from this stage
            app.Lock.Lock()
            if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second {
                if len(app.TaskInformation[args.Stage]) > 1 {
                    app.LogFileChan <- fmt.Sprintf("Downscaling Stage: %d Task: %d Rate: %.2f\n
                    fmt.Printf("Downscaling Stage: %d Task: %d Rate: %.2f\n", args.Stage, args.
                    app.removeTask(args.Stage)
                    // Reset the timer
                    app.LastAutoscaleTime[args.Stage] = time.Now()
                }
            }
            app.Lock.Unlock()
        } else if args.Rate > app.HighestRate {
            // add a task to this stage
            app.Lock.Lock()
            if time.Since(app.LastAutoscaleTime[args.Stage]) > 3*time.Second {
                taskNum := app.NextTaskNum[args.Stage]
                app.NextTaskNum[args.Stage]++
                app.LogFileChan <- fmt.Sprintf("Upscaling Stage: %d Rate: %.2f\n", args.Stage,
                app.addTask(args.Stage, taskNum)
                app.sendIps()
                // Reset the timer
                app.LastAutoscaleTime[args.Stage] = time.Now()
            }
            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 o
        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",
    }
    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
    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()

    // FIX: Add Timeout so we don't block forever if a worker is slow/dead
    timeout := time.After(2 * time.Second)
    for i := 0; i < numSuccess; i++ {
        select {
        case x := <-waitingChan:
            if x.Error != nil {
                fmt.Println("Failed to send IPs to workers: " + x.Error.Error())
            }
        case <-timeout:
            fmt.Println("Warning: Timeout waiting for workers to update IPs. Continuing...")
            return // Stop waiting and release the lock
        }
    }
}

func (app *RainStorm) initWorker() { // MUST BE CALLED INSIDE RAINSTORM LOCK --> only calle
    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 C
    //if taskNum > app.StageCounter[stageNum]) {
    // app.TaskInformation[stageNum] = append(app.TaskInformation[stageNum], workers.ips[ap
    //} else {
```

```
    // app.TaskInformation[stageNum][taskNum] = workers.ips[app.NextAvailableVM%numWorkers]
    //}
    workers.l.RLock()
    app.TaskInformation[stageNum][taskNum] = &TaskInfo{Ip: workers.ips[app.NextAvailableVM%
    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.Task
}

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()

    // FIX: Send Async! Do not wait for the worker to cleanup.
    // Use .Go instead of .Call so we don't hold the lock if the worker is busy.
    _ = rpcWorker.Go("Worker.AutoscaleDown", task, &reply, nil)
    // err check is moot because it's async, we assume success or retry later.
}

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++
                  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":
   //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
   vm := splits[1]
```

```
            pid, _ := strconv.Atoi(splits[2])

            // We need to release the lock before calling ReceiveFailure to avoid Deadlock,
            // because ReceiveFailure acquires the lock itself.
            var foundTask bool
            var targetTask Task

            curApp.Lock.RLock()
            for stageNum, stage := range curApp.TaskInformation {
               for taskNum, info := range stage {
                  if vm == info.Ip.String() && pid == info.Pid {
                     targetTask = Task{
                        TaskNumber: taskNum,
                        Stage:      stageNum,
                     }
                     foundTask = true
                     break
                  }
               }
               if foundTask {
                  break
               }
            }
            curApp.Lock.RUnlock()

            if foundTask {
               // 1. Kill the physical process on the worker
               rpcWorkersLock.RLock()
               worker, ok := rpcWorkers[vm] // Use the vm string to find the client
               rpcWorkersLock.RUnlock()

               if ok {
                  var reply int
                  err := worker.Call("Worker.KillTask", TaskID{
                     Task:  targetTask.TaskNumber,
                     Stage: targetTask.Stage,
                  }, &reply)

                  if err != nil {
                     fmt.Println("Error killing task:", err)
                  } else {
                     fmt.Println("Task Killed via RPC.")
                  }
               }

               // 2. IMPORTANT: Tell the Leader to recover/reschedule immediately
```

```
                fmt.Println("Triggering Leader Recovery...")
                var reply int
                // We call ReceiveFailure locally to update the map and send new IPs to everyo
                curApp.ReceiveFailure(targetTask, &reply)
            } else {
                fmt.Println("Task not found to kill.")
            }
            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

        }
    }
}
```

**User**

I think the problem is that I am sending a task and stage to the worker but for some reason it is getting a different task and stage number

**Gemini**

This is a classic "RPC Serialization Mismatch." You are absolutely right—the Worker is receiving the wrong numbers (specifically `0`) because the **field names** in the structs sent by the Leader do not match the field names expected by the Worker.

### The Problem: `TaskNumber` vs. `Task`

When using Go's `net/rpc`, the data is encoded using `gob`. For a struct to be successfully decoded on the other side, the **field names must match exactly**.

1. **Leader sends `Task` struct:**

   Go

   ```go
   type Task struct {
       TaskNumber int  // <--- Field Name is "TaskNumber"
       Stage      int
       Executable Operation
   }
   ```

2. **Worker expects `TaskID` struct:**

   Go

   ```go
   type TaskID struct {
       Task  int       // <--- Field Name is "Task"
       Stage int
   }
   ```

**The Result:**

- The `Stage` fields match, so the Stage number is transferred correctly.
- **The Mismatch:** The Leader sends `TaskNumber`, but the Worker is looking for `Task`. The `gob` decoder ignores `TaskNumber` and sets the Worker's `Task` field to its default value: **0**.
- **The Bug:** You try to kill Task 5, but the Worker receives "Kill Task 0".

### The Fix

In your Leader's `removeTask` function, you must convert the `Task` object into a `TaskID` object (or whatever struct the Worker expects) before sending the RPC.

**Update `removeTask` in `main.go`:**

Go

```
func (app *RainStorm) removeTask(stageNum int) { // MUST BE WRAPPED IN APP 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 {
       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()

    // --- FIX IS HERE ---
    // Create the struct that MATCHES the Worker's AutoscaleDown argument exactly
    taskID := TaskID{
       Task:  taskNum, // Map 'TaskNumber' to 'Task'
       Stage: stageNum,
    }

    var reply int
    rpcWorkersLock.RLock()
    rpcWorker := rpcWorkers[deletedTaskIp.Ip.String()]
    rpcWorkersLock.RUnlock()

    // Send 'taskID', NOT 'task'
    fmt.Printf("Leader requesting kill for Stage: %d, Task: %d\n", taskID.Stage, taskID.Tas
    _ = rpcWorker.Go("Worker.AutoscaleDown", taskID, &reply, nil)
}
```

## Double Check `kill_task`

You likely have the same issue in your `processStdin` function where you handle the manual `kill_task` command. Check lines ~630 in your previous code:

Go

```go
// Inside kill_task case in processStdin:

// OLD (Wrong):
// targetTask is likely type 'Task' (TaskNumber)
err := worker.Call("Worker.KillTask", TaskID{
    Task:  targetTask.TaskNumber, // Ensure this maps correctly!
    Stage: targetTask.Stage,
}, &reply)
```

If you were manually constructing `TaskID` there, it was probably working fine. The issue was specifically in `removeTask` where you were passing the full `Task` struct directly to a function expecting `TaskID`.

**User**

the worker is also not getting the stage -1 completion when the file finished reading

**Gemini**

The reason the workers are not receiving the "Stage -1 Completion" signal is likely that your **Leader is getting stuck waiting for a response from a "Zombie" or slow worker.**

In your `sendStageCompletion` function, you wait for **every single worker** to acknowledge the message.

Go

```go
// CURRENT BLOCKING LOGIC
for i := 0; i < numSuccess; i++ {
   x := <-waitingChan // <--- BLOCKS FOREVER if one worker doesn't reply
   // ...
}
```

If just one worker has crashed, failed to downscale correctly, or is simply overwhelmed, the Leader will hang on this line forever. It will never finish broadcasting the message, and it will stop processing anything else.

### The Fix

You need to add a **Timeout** to `sendStageCompletion`, exactly like we did for `sendIps`. This ensures that even if one worker is unresponsive, the Leader moves on and notifies the others.

**Update `sendStageCompletion` in your Leader ( `main.go` ):**

Go

```go
func (app *RainStorm) sendStageCompletion(completedStage int) {
    waitingChan := make(chan *rpc.Call, len(rpcWorkers))
    numSuccess := 0
    rpcWorkersLock.RLock()
    for _, worker := range rpcWorkers {
       var reply int
       // Use .Go for async sending
       worker.Go("Worker.ReceiveFinishedStage", completedStage, &reply, waitingChan)
       numSuccess++
    }
    rpcWorkersLock.RUnlock()

    // --- FIX: Add Timeout so we don't hang on a dead worker ---
    timeout := time.After(2 * time.Second)
    for i := 0; i < numSuccess; i++ {
       select {
       case x := <-waitingChan:
          if x.Error != nil {
             fmt.Println("Failed to send completed stageID to workers: " + x.Error.Error())
```

```
            }
        case <-timeout:
            fmt.Printf("Warning: Timeout waiting for workers to ACK Stage %d completion. Movi
            return // Stop waiting, release the Leader to do other things
        }
    }
}
```

### Why this happens specifically with Stage -1

1. The Input Reader finishes the file.
2. It calls `sendStageCompletion(-1)`.
3. The Leader sends the RPC to all workers.
4. **Scenario:** One worker might be stuck cleaning up a task from a previous `AutoscaleDown` event or is overloaded.
5. The Leader waits for that specific worker's reply.
6. The Leader blocks.
7. Since the Leader is blocked, it can't handle any other tasks, effectively freezing the system state.

By adding the timeout, you ensure the signal is "Fire and Forget" (mostly). The healthy workers will receive the signal and close their pipes, allowing the job to finish.