gemini-conversation

Exported on: 12/7/2025, 10:24:33 PM

User

Ok here is my leader and client for the rainstorm processing respectively, Can you help me understand why a lot of final tuples are not being sent to the leader, even though are being processed at the worker?

Gemini

Based on the `main.go` file provided, the issue where final tuples are processed by the worker but not appearing in your leader's output is likely due to a **Race Condition during the application shutdown sequence**.

Here is the breakdown of the problem and the solution:

The Root Cause: Premature `cancel()`

When the last worker finishes, it sends a `ReceiveTaskCompletion` RPC to the Leader. This triggers the following chain of events in your Leader:

1. `ReceiveTaskCompletion` detects the job is done and sends `true` to `appCompletedChan`.
2. The `main()` function unblocks at `<-appCompletedChan`.
3. **Immediately**, `main()` calls `cancel()`.
4. The `cancel()` signal propagates to the `tupleListener` goroutines (handling the incoming data) via `ctx.Done()`.
5. **The Critical Failure**: The goroutines reading from the socket stop **instantly**, dropping any data that is currently buffered in the TCP socket or the `bufio.Reader` but hasn't been processed yet.

Visualizing the Race:

Plaintext

```
[Worker]                [Network]               [Leader]
   |                        |                      |
   |-- (1) Send Tuple data ->|-> (Buffered)         |
   |                        |                      |
   |-- (2) Send Finish RPC ->|-> (Arrives Fast) ----> | -> Triggers appCompletedChan
                                                   | -> calls cancel()
                                                   | -> Tuple Reader receives ctx.Done()
                                                   | -> RETURNS/QUITS (Tuple data lost)
```

Secondary Cause: Channel Blocking

Your `outputChan` is buffered to `200`. If your HyDFS write (which happens in the consumer goroutine) is slower than the incoming rate of tuples, the `outputChan` will fill up. This will cause the `tupleListener` readers to block. If the shutdown sequence initiates while they are blocked, or if the workers time out, you will lose data.

Solutions

1. Graceful Shutdown (High Priority Fix)

You need to ensure the Leader finishes reading from the network before cancelling the context. Since the workers are "done," they should effectively close their TCP connections to the Leader. You should wait for these connections to close or drain.

Modified `main` logic: Don't call `cancel()` immediately. Instead, allow the HyDFS writer to drain the channel, and give the network readers a moment to finish.

Go

```
// Inside main()

<-appCompletedChan // All tasks reported "Done"
println("RainStorm Application completed!")

// FIX: Give time for in-flight tuples to arrive and be processed
// A simple sleep is a "hacky" but effective fix for this specific race
time.Sleep(2 * time.Second)

// Better Fix: Close the listener so no NEW connections come in,
// but let existing readers finish.
_ = tupleListener.Close()

// Ideally, you would use a WaitGroup here to wait for all reader
// connections to return EOF, but since you don't track them,
// a sleep + channel drain is the immediate fix.

cancel() // NOW cancel the context to stop the writers
close(r.LogFileChan)
```

2. Ensure HyDFS Writer Flushes on Shutdown

Your HyDFS writer handles `ctx.Done()` by checking `buffer.Len() > 0`. This is good, but it relies on `outputChan` being empty.

If you use the `time.Sleep` method above, the `outputChan` will naturally drain because the consumer is still running (Context hasn't been cancelled yet).

3. Worker-Side Acknowledgment (Best Practice)

In your Worker code (not provided, but implied), ensure you are **closing the TCP connection** to the Leader *before* sending the `ReceiveTaskCompletion` RPC.

If the Worker keeps the connection open, the Leader doesn't know if more data is coming.

- **Worker:** Process Tuple -> Send Tuple -> Close Tuple Connection -> Send RPC "Task Done".
- **Leader:** `ReadString` returns `EOF` (error). You can catch this and exit that specific reader goroutine cleanly.

Summary Checklist for Fix

1. **Leader (`main.go`)**: Insert a `time.Sleep(time.Second * 2)` after `<-appCompletedChan` and before `cancel()`. This allows the TCP buffers to be read and the `outputChan` to be written to HyDFS.
2. **Leader (`main.go`)**: Increase `outputChan` buffer size (e.g., to `1000`) to prevent temporary lags in HyDFS from blocking the network readers.
3. **Worker**: Ensure the connection sending tuples to the leader is flushed/closed before the worker signals completion via RPC.

User

How can I use the wait group to make sure all the reader connections are done, I want to do that because it is correct

Gemini

Using a `sync.WaitGroup` is indeed the correct and robust way to handle this. It allows your Leader to explicitly track every active network connection and ensures that `main()` does not exit until every byte has been read and processed.

Here is how to modify your `main.go` to implement this.

Step 1: Initialize the WaitGroup

Inside your main application loop (inside `for { r := <-input ... }`), define a `WaitGroup` to track open connections.

Go

```go
// ... inside main loop ...
r := <-input
if numWorkers == 0 {
    panic("No workers")
}

// 1. Create the WaitGroup here
var connWg sync.WaitGroup

r.LogFileChan = make(chan string, 100)
// ...
```

Step 2: Track Connections in the Listener

Modify your `tupleListener` goroutine. You must increment the counter **before** launching the handler goroutine and decrement it using `defer` **inside** the handler.

Find this block in your code:

Go

```go
// listen for tuples to print to console and buffered append to hydfs
go func() {
    for {
        select {
        case <-ctx.Done():
            _ = tupleListener.Close()
            return
        default:
            conn, err := tupleListener.Accept()
            if err != nil {
```

```
                continue
            }

            // 2. Increment the WaitGroup for every new connection
            connWg.Add(1)

            go func(conn net.Conn) {
                // 3. Decrement when this connection is fully done/closed
                defer connWg.Done()
                defer conn.Close()

                reader := bufio.NewReader(conn)
                for {
                    select {
                    case <-ctx.Done():
                        return
                    default:
                        // ... reading logic ...
                    }
                }
            }(conn)
        }
    }
}()
```

Step 3: Update the Shutdown Sequence

This is the most critical part. Instead of cancelling immediately, you must close the listener (to stop *new* connections) and then wait for the `connWg` (to finish *current* connections).

Find the cleanup block at the end of the loop and change it to this:

Go

```
// ...
<-appCompletedChan // Wait for workers to say "Job Done"
println("RainStorm Application completed!")

// 4. Close the listener first.
// This stops the Accept() loop, so no NEW connections can be added to the group.
_ = tupleListener.Close()

// 5. Wait for all active connections to finish reading their data.
// The Leader will pause here until every worker closes their socket.
connWg.Wait()

// 6. NOW it is safe to cancel the context.
```

```
// All data has been read and put into outputChan.
cancel()

// ... remaining cleanup code ...
```

Why this fixes the bug

1. **Isolation:** Even though `appCompletedChan` fires, `ctx.Done()` is not triggered yet. This means your reader goroutines (`case <-ctx.Done():`) keep running.
2. **Draining:** The readers will continue to loop until `reader.ReadString` returns an EOF error (which happens when the Worker closes the socket).
3. **Completion:** Once all readers hit EOF and exit, `connWg` hits zero. The `Wait()` releases, and only *then* do you shut down the file writers and logger.

User

Ok, thanks that fixed this, now when I try to run 2 rainstorm jobs from the same leader, one after another, it gets stuck. I want to know if it based on the client or the leader, for now I am asking you to look at the client. Here is the output after 2 jobs from the leader:
Started Rainstorm task told leader im done 0-3 told leader im done 0-7 told leader im done 0-0 told leader im done 0-1 told leader im done 0-4 told leader im done 0-8 told leader im done 0-5 told leader im done 0-9 leader responded 0-3 told leader im done 0-2 leader responded 0-7 told leader im done 0-6 leader responded 0-0 leader responded 0-1 leader responded 0-4 leader responded 0-8 leader responded 0-5 leader responded 0-9 leader responded 0-2 This job finished 2025/12/07 14:01:38 rpc.Serve: accept:accept tcp [::]:8021: use of closed network connection read tcp 172.22.94.231:60900->172.22.158.45:8023: use of closed network connection leader responded 0-6 Started Rainstorm task

Gemini

Based on the code provided in `main.go`, the issue where your second job "gets stuck" is caused by a **Resource Leak (Port Conflict)** on the `TuplePort`.

The specific problem is a "Zombie Listener." When your first job finishes, you close the `leaderListener` and the `rainStormLeader` connection, but you **never close the `tupleListener`** (the TCP listener handling tuple transfers between workers).

The Mechanics of the Bug

1. **Job 1 Starts:** You launch a goroutine (Line 138) that calls `net.Listen("tcp", TuplePort)`. It sits in a `for` loop accepting connections.
2. **Job 1 Ends:** The `<-worker.done` signal triggers. You close `leaderListener` and `tupleSendConn` (Line 287-289), but the "reading tuples" goroutine is still running, holding the `TuplePort` open.
3. **Job 2 Starts:** The `main` loop restarts. A new goroutine tries to call `net.Listen("tcp", TuplePort)` (Line 139).
4. **The Failure:** Because the Job 1 goroutine is still holding the port, `net.Listen` returns an error ("Address already in use"). Your code checks `if err != nil { return }` (Line 141), causing the **Job 2 tuple listener to exit immediately**.
5. **The "Stuck" State:**
    - Job 2 tasks start successfully, but the Worker has no listener to receive data.
    - Any tuples sent to this machine are accepted by the **Job 1 Listener** (which is still running).
    - Job 1's listener tries to send data to Job 1's tasks (which are deleted), so the data is lost/dropped.
    - Job 2 sits idle waiting for data that never arrives.

The Fix

You must lift the `tupleListener` creation out of the anonymous goroutine so that `main()` can control it and explicitly close it during cleanup.

Here is the corrected logic for your `main` loop:

Go

```
// Inside the main loop, before launching goroutines

// 1. Create the listener here (Blocking main from continuing if port is stuck)
```

```
tupleListener, err := net.Listen("tcp", TuplePort)
if err != nil {
    // If we hit this, the OS hasn't released the port yet or a zombie exists
    fmt.Println("Error listening on TuplePort:", err)
    time.Sleep(1 * time.Second)
    continue
}

// 2. Goroutine for reading in tuples
go func(listener net.Listener) { // Pass listener as argument
    // REMOVE the net.Listen call from here

    // Defer close is okay here for safety, but main will handle the real close
    defer func() {
        _ = listener.Close()
    }()

    for {
        conn, err := listener.Accept() // Use the passed listener
        if err != nil {
            return // This will happen when main closes the listener
        }
        go func(conn net.Conn) {
             // ... existing tuple processing logic ...
        }(conn)
    }
}(tupleListener)

// ... (Other goroutines: Sending tuples, Resource Manager, Logging) ...

<-worker.done
println("This job finished")

// 3. Cleanup Phase: EXPLICITLY close the tuple listener
_ = tupleListener.Close() // This breaks the Accept() loop in the goroutine above

_ = leaderListener.Close()
_ = worker.rainStormLeader.Close()
_ = worker.tupleSendConn.Close()
time.Sleep(1 * time.Second)
```

Why the error logs you see are "Normal"

You mentioned seeing: `rpc.Serve: accept:accept tcp [::]:8021: use of closed network connection`.

This is actually expected behavior and **not** the cause of the stuck job.

- **Cause:** This happens when you call `leaderListener.Close()` at the end of a job.

- **Effect:** The RPC server (running in `go server.Accept(leaderListener)`) tries to accept a connection on a closed socket, fails, and prints that error. It is harmless noise during shutdown.

Summary Checklist

1. **Move `net.Listen`:** Call it in `main` scope, not inside the goroutine.
2. **Pass the Listener:** Pass the resulting `net.Listener` variable into the reading goroutine.
3. **Close in Cleanup:** Add `_ = tupleListener.Close()` to the block after `<-worker.done`.

User

Ok, I did that, I have a question though, should I be using wait groups and cancellation contexts to close these gorountines in the worker?

Gemini

Yes, you absolutely should. In fact, looking at your `Worker` code, **you currently have a critical resource leak.**

Because your `main()` function runs in a loop (restarting the `Worker` struct for every new job), any goroutine you start that doesn't explicitly exit will become a **"Zombie Goroutine."** It will keep running in the background while the new job starts.

The Danger: Zombie Accumulation

After running 3 jobs, you will have:

- 3 **Resource Managers** (all trying to lock mutexes and send RPCs).
- 3 **Log Flushers** (all writing to HyDFS).
- 3 **Tuple Senders** (waiting on channels that will never be written to).

This will eventually crash your program or cause massive race conditions.

Here is how to apply `sync.WaitGroup` and `context.Context` to fix this.

Step 1: Setup Context and WaitGroup

Inside your `main` loop, before starting any goroutines:

Go

```
// Inside main(), inside the for loop
worker := Worker{ ... } // existing code

// 1. Create a Context to signal shutdown
ctx, cancel := context.WithCancel(context.Background())

// 2. Create a WaitGroup to track background routines
var wg sync.WaitGroup
```

Step 2: Fix the Resource Manager (The Critical Fix)

Your Resource Manager currently loops forever on a ticker. It needs to stop when `ctx.Done()` fires.

Go

```
// Increment WG
wg.Add(1)
go func() {
    defer wg.Done() // Decrement when exiting
```

```
    // Use NewTicker so we can stop it cleanly
    ticker := time.NewTicker(time.Second)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done(): // Stop signal received
            return
        case <-ticker.C:
            // ... Your existing Resource Manager logic ...
            worker.tasksLocker.RLock()
            // ...
            worker.tasksLocker.RUnlock()
        }
    }
}()
```

Step 3: Fix the Log Flusher

This also runs on a ticker and needs the same treatment.

Go

```
wg.Add(1)
go func() {
    defer wg.Done()

    // Use NewTicker
    ticker := time.NewTicker(500 * time.Millisecond)
    defer ticker.Stop()

    // ... setup buffers ...

    for {
        select {
        case <-ctx.Done():
            // Optional: Flush remaining buffers one last time here
            return
        case <-ticker.C:
             // ... existing flush logic ...
        case req := <-worker.logChan:
             // ... existing log logic ...
        }
    }
}()
```

Step 4: Fix the Tuple Sender

This goroutine currently blocks on `<-worker.taskOutputs`. If the job ends, it sits there forever.

Go

```go
wg.Add(1)
go func() {
    defer wg.Done()
    for {
        select {
        case <-ctx.Done():
            return
        case out := <-worker.taskOutputs:
            // ... All your existing tuple sending logic ...
            // ... Next Stage logic ...
        }
    }
}()
```

Step 5: Update the Cleanup Sequence

Now, when the job finishes, you signal everyone to stop and wait for them to die before restarting the loop.

Go

```go
// ... inside main ...
<-worker.done
println("This job finished")

// 1. Signal all background goroutines to stop
cancel()

// 2. Close network listeners (breaks the Accept() loops)
_ = leaderListener.Close()
_ = worker.rainStormLeader.Close()
_ = worker.tupleSendConn.Close()
_ = tupleListener.Close()

// 3. Wait for the Resource Manager, Logger, and Sender to actually exit
wg.Wait()

// 4. Now safe to restart the loop or exit
time.Sleep(1 * time.Second)
```

Summary of Benefits

1. **No Leaks:** The `wg.Wait()` guarantees that when `main` loops back to the top, the previous worker is 100% gone.
2. **No Race Conditions:** You won't have an old Resource Manager trying to read `worker.tasks` (which belongs to the old worker) while the new worker is initializing.
3. **Clean Logs:** You ensure your file buffers are flushed before the program moves on.

User

I did, is this good ?

Gemini

You are very close, but there is one **critical concurrency bug** in how you are using `wg.Add(1)`.

The Bug: `wg.Add(1)` inside the Goroutine

In your code, you are doing this:

Go

```
go func() {
    wg.Add(1) // <--- THIS IS DANGEROUS
    defer wg.Done()
    // ...
}()
```

Why it fails: Go scheduling is non-deterministic. It is possible that your `main` thread reaches `wg.Wait()` (at the end of the function) **before** the new goroutine has actually started and called `wg.Add(1)`. If that happens, `wg.Wait()` sees a counter of `0`, returns immediately, and your program restarts the loop while the old goroutine is still running (creating a zombie).

The Fix: You **must** call `wg.Add(1)` **before** the `go` keyword.

Corrected Code Blocks

Here are the specific fixes for your `main.go`.

1. Fix Tuple Sender (Line 130)

Change this:

Go

```
// BAD:
// go func() {
//    wg.Add(1)
//    ...

// GOOD:
wg.Add(1) // Call this BEFORE 'go'
go func() {
    defer wg.Done()
    for {
        // ... (rest of logic is fine)
```

2. Fix Tuple Listener (Line 252)

You forgot to add the listener goroutine to the WaitGroup! If you don't wait for this, you might try to `net.Listen` in the next job before this one fully shuts down.

Change this:

Go

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

wg.Add(1) // <--- Add this
go func(listener net.Listener) {
    defer wg.Done() // <--- Add this
    for {
        conn, err := listener.Accept()
        // ...
```

3. Fix Resource Manager (Line 331)

Change this:

Go

```
wg.Add(1) // <--- Move this outside
go func() {
    defer wg.Done()
    ticker := time.NewTicker(time.Second)
    // ...
```

4. Fix Log Flusher (Line 372)

Change this:

Go

```
wg.Add(1) // <--- Move this outside
go func() {
    defer wg.Done()
    // ...
```

Summary of Cleanup

Your cleanup block at the end looks correct, assuming you make the changes above.

Go

```
<-worker.done
println("This job finished")

cancel() // Signal everyone to stop

// Close listeners to break the "Accept" loops
_ = leaderListener.Close()
_ = worker.rainStormLeader.Close()
_ = worker.tupleSendConn.Close()
_ = tupleListener.Close()

wg.Wait() // Now this will ACTUALLY wait for everyone to finish
time.Sleep(1 * time.Second)
```

User

Ok , changed that, everything else is good?

Gemini

This is looking much better. The structure is now safe against the "Zombie Worker" issues we discussed.

However, there is **one remaining critical bug** in your `tupleListener` logic that will cause your Worker to **panic (crash)** if a tuple arrives slightly late (after a task has finished).

1. The "Late Tuple" Panic

In your tuple reader goroutine (lines 282-288), you access `worker.tasks[targetTask]` directly. If the task has already finished and been deleted from the map (which happens in your `AddTask` closure), accessing it will return `nil`. Trying to access `.input` on a `nil` pointer causes a panic.

Fix: Check if the task exists before writing to it.

Go

```
// inside tupleListener goroutine...

worker.tasksLocker.Lock()
task, exists := worker.tasks[targetTask] // 1. Check existence
if !exists {
    worker.tasksLocker.Unlock()
    // Task is gone (finished/failed).
    // We should probably just ACK it so the sender stops retrying, or drop it.
    // Sending ACK to prevent sender from hanging/retrying forever:
    ackMsg := fmt.Sprintf("%s-%s\n", split[0], ACK)
    _, _ = fmt.Fprintf(conn, ackMsg)
    continue
}

_, err = io.WriteString(task.input, split[3]) // 2. Use the safe 'task' variable
if task.inputRate == 0 {
    task.startTime = time.Now()
}
task.inputRate++
worker.tasksLocker.Unlock()
```

2. Dead Code Clean-up

At the start of your loop (lines 61-64), you have a logic error that is technically harmless but confusing:

Go

```
server := rpc.NewServer()
if err != nil { // 'err' here is nil from the previous success, or stale
    time.Sleep(1 * time.Second)
```

```
    continue
}
```

`rpc.NewServer()` does not return an error. You can delete lines 62-65.

3. Connection Safety (Context Monitoring)

In your inner `tupleListener` loop (Line 259), you loop forever on `reader.ReadString`. If a sender worker keeps the connection open but stops sending data during a shutdown, this goroutine will sit there until the OS kills the socket.

To make this 100% robust, you can set a read deadline or check the context. **Simple Fix:** Just add a read deadline so it checks for shutdown periodically.

Go

```go
// Inside the inner connection loop
for {
    // Check for shutdown signal before reading
    select {
    case <-ctx.Done():
        return
    default:
    }

    // Set a deadline so ReadString doesn't block forever if the network hangs
    _ = conn.SetReadDeadline(time.Now().Add(5 * time.Second))
    tuple, err := reader.ReadString('\n')

    if err != nil {
        // Check if it was a timeout
        if netErr, ok := err.(net.Error); ok && netErr.Timeout() {
            continue // Loop back and check ctx.Done()
        }
        return // Actual error or EOF
    }
    // ... process tuple ...
}
```

Summary

If you apply the **Panic Fix (#1)**, your code is "good to go" and should be stable. Fix #3 is nice-to-have for extreme edge cases, but #1 is mandatory to prevent crashes.

User

Can you help me understand why I am getting the kill task but then I am not sending to the leader that its dead? Here is the output of the worker:
Started Rainstorm task Received kill task Killing task: &{0xc0001a1080 0xc000048478 0xc000048480 88 {13998764776145309197 5816224892 0x9cfc00} {13998764790367189622 19005719778 0x9cfc00} 85 0xc0000484b0} Scanner error: read |0: file already closed Didn't receive ack, retrying

Gemini

Based on the `main.go` file and the output provided, the reason your worker is not reporting the "death" to the leader is that **your monitoring goroutine exits too early.**

When `KillTask` is called, it explicitly closes the `task.output` (the stdout pipe). This causes the `bufio.Scanner` in your `AddTask` goroutine to fail immediately.

The Problem

Look at this section in your `AddTask` function (around lines 361-364):

Go

```
            if scanner.Err() != nil {
                    fmt.Println("Scanner error:", scanner.Err())
                    return // <--- THIS IS THE CULPRIT
            }

    // The code below is NEVER reached if scanner.Err() occurs
            err = cmd.Wait()
            if err == nil {
        // ... Report Success ...
            } else {
        // ... Report Failure ...
                    var reply int
                    _ = w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &reply)
                    fmt.Printf("Task %v failed: %v\n", t, err)
            }
```

When you kill the task:

1. `KillTask` closes the pipe.
2. `scanner.Scan()` returns `false`.
3. `scanner.Err()` returns `"read |0: file already closed"`.
4. Your code prints the error and **returns immediately**.
5. It **skips** `cmd.Wait()` and the subsequent `ReceiveFailure` RPC call.

The Fix

You simply need to **remove the `return` statement** (or check for that specific error) so the code can proceed to wait for the process to exit and notify the leader.

Modify the goroutine in `AddTask` like this:

Go

```
	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())
			// DON'T RETURN HERE. Let the code proceed to cmd.Wait()
		}

		// Now this code will actually run!
		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)
		}

	// ... cleanup logic ...
```

User

Can you explain why my worker is no sending any task completions if a task fails and is restarted on the same worker by the leader?

Gemini

Based on the code provided in `main.go`, the reason your restarted task never sends a completion signal is that **the Worker does not "remember" that the previous stage has already finished.**

Here is the sequence of events leading to the bug:

1. **Stage Completion (Initially):** The previous stage finishes. The Leader sends `ReceiveFinishedStage` to your worker. Your worker iterates over the *currently running* tasks and closes their input pipes (Lines 393-397). This allows those tasks to finish reading and exit.
2. **Task Failure:** One of your tasks (let's call it Task A) fails *after* this point (or during processing). The old process for Task A is gone.
3. **Task Restart (`AddTask`):** The Leader calls `AddTask` to restart Task A. `AddTask` creates a **new** `localTask` with a **new** `input` pipe (Line 424: `taskStdin`). It handles recovery (writing old tuples from logs), but acts as if the stream is still open.
4. **The Deadlock:** Since the `ReceiveFinishedStage` RPC was already received and processed in the past, **it will not be sent again.** Your worker has no state variable recording that "Stage X is finished." Therefore, the new task's input pipe remains open indefinitely. The task process sits waiting for input (EOF) that will never come, so `cmd.Wait()` never returns, and `ReceiveTaskCompletion` is never called.

The Fix

You need to add state to your `Worker` struct to track finished stages, and check this state when adding a new task.

Step 1: Update `Worker` struct Add a map to track finished stages.

Go

```go
type Worker struct {
    // ... existing fields ...
    finishedStages map[int]bool // Add this field
}
```

Step 2: Initialize the map Initialize it in `main()` inside the worker creation loop.

Go

```go
worker := Worker{
    // ...
    finishedStages: make(map[int]bool), // Initialize here
}
```

Step 3: Update `ReceiveFinishedStage` Record the stage completion.

Go

```go
func (w *Worker) ReceiveFinishedStage(stage int, reply *int) error {
    w.tasksLocker.Lock() // Use Lock (Write) instead of RLock
    w.finishedStages[stage] = true // Remember this!

    // ... existing logic to close CURRENT tasks ...
    var inputsToClose []io.WriteCloser
    for key, value := range w.tasks {
        if key.Stage == stage+1 {
            inputsToClose = append(inputsToClose, value.input)
        }
    }
    w.tasksLocker.Unlock() // Unlock

    // ... existing loop to close inputs ...
    return nil
}
```

Step 4: Update `AddTask` Check if the previous stage is already done. If so, close the input pipe **after** writing recovered tuples.

Go

```go
// Inside AddTask, AFTER the recovery logic (Lines 464-471)
// and just before returning.

    // ... (existing recovery loop) ...
        for _, tuple := range tuples {
            _, err = io.WriteString(taskStdin, tuple+"\n")
            // ...
        }
    } // End of recovery block

    // NEW LOGIC: Check if we missed the finish signal
    w.tasksLocker.RLock()
    isPrevStageFinished := w.finishedStages[t.Stage-1]
    w.tasksLocker.RUnlock()

    if isPrevStageFinished {
        // The previous stage is already done, so this task
        // should not expect any NEW data from the network.
        // We close the input so it processes recovered data and then exits.
        _ = taskStdin.Close()
    }

    *reply = task.Process.Pid
```

```
    return nil
}
```

User

Ok, but no task completions at all are being sent to the leader, here is the output from my worker:
Started Rainstorm task received new IPSreceived new IPSReceived kill task Killing task: &{0xc000203080 0xc000136778 0xc000136780 106 {13998767762373922721 29487703749 0x9cfc00} {13998767771561901715 38012006317 0x9cfc00} 97 0xc0001367b0} Scanner error: read |0: file already closed Task {1 8} failed: signal: killed Received stage -1 completed

Gemini

Based on the logs you provided, your worker **is attempting** to tell the Leader that the task failed, but you are likely suppressing an error that explains why it's not working.

Here is the log line proving the code reached the failure block: `Task {1 8} failed: signal: killed`

This print statement is inside the `else` block of your `AddTask` goroutine, right after this line:

Go

```
_ = w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &reply)
```

Because you use `_ =`, if the RPC fails (e.g., network timeout, connection closed), you fail silently.

1. Immediate Debugging Fix

Change the ignored error to a print statement so you can see **why** the RPC is failing.

In `main.go`, inside `AddTask`:

Go

```
// Inside the else block for cmd.Wait() error
var reply int
err = w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &reply) // Capture err
if err != nil {
    fmt.Println("Error sending ReceiveFailure RPC:", err)
}
fmt.Printf("Task %v failed: %v\n", t, err)
```

2. The Logical Bug: Reporting "Failure" on Intentional Kill

There is a logic flaw in your synchronization. When the Leader calls `KillTask`, it **knows** the task is dying. However, your worker treats this "Kill" as a crash and reports it as a `ReceiveFailure`.

This can confuse the Leader (causing it to restart the task redundantly or ignore the message because it thinks the task is already gone).

Fix: Check if the task was removed from your `tasks` map **before** reporting failure. `KillTask` removes the task from the map immediately, so if it's gone, you know it was an intentional kill.

Modified `AddTask` logic:

Go

```
// ... inside the goroutine ...
err = cmd.Wait()

// Check if task was intentionally killed (removed from map)
w.tasksLocker.RLock()
_, stillActive := w.tasks[tId]
w.tasksLocker.RUnlock()

if err == nil {
    // ... Success logic ...
} else {
    // ONLY report failure if the task is still supposed to be running
    if stillActive {
        var reply int
        errRPC := w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &reply)
        if errRPC != nil {
            fmt.Println("RPC Failed:", errRPC)
        }
        fmt.Printf("Task %v failed: %v\n", t, err)
    } else {
        fmt.Printf("Task %v killed intentionally (no failure report sent)\n", t)
    }
}
// ...
```

3. The "Restart" Deadlock (Crucial)

If the Leader **restarts** this task on the same worker, it will likely **hang forever** because of the state issue we discussed previously.

When a task fails and restarts, the new task waits for input. But since `ReceiveFinishedStage` was already processed for the previous stage, the worker doesn't know it needs to close the input pipe for the *new* task.

You must implement the `finishedStages` fix I detailed in the previous response:

1. Add `finishedStages map[int]bool` to `Worker`.
2. Update it in `ReceiveFinishedStage`.
3. Check it in `AddTask` and close the input pipe immediately if the previous stage is already done.

User

Ok, no I know the leader is getting the failure rpc because it is logged on the leader. Also the failed task in in stage 1, not stage 0, so atleast stage 0 should be completed no? I agree that stage 1 might hang.

Gemini

You are absolutely correct. **Stage 0 has likely completed**, and that is exactly *why* your restarted Stage 1 task is hanging.

This is a classic "Lost Signal" bug caused by the timing of the restart.

The Sequence of Events Causing the Hang

1. **Stage 0 Completes:** All Stage 0 tasks finish. The Leader sends `ReceiveFinishedStage(0)` to your Worker.
2. **Worker Updates State:** Your Worker receives this RPC and closes the inputs for all *currently running* Stage 1 tasks.
3. **Task Failure:** Sometime later (or concurrently), your specific Stage 1 task (Task 8) fails and is killed.
4. **Task Restart (`AddTask`):** The Leader restarts Task 8 by calling `AddTask`.
5. **The Bug:**
   - `AddTask` starts the new process and opens a **new** input pipe (`stdin`).
   - The Worker **does not check** if Stage 0 is already done.
   - Because the `ReceiveFinishedStage(0)` RPC was *already* received in the past, it will **never be sent again**.
   - The new Task 8 sits there waiting for input from Stage 0. Since Stage 0 is done, no data ever comes, but the pipe remains open.
   - Task 8 never exits `scanner.Scan()`, never calls `cmd.Wait()`, and never sends a completion RPC.

The Solution: "Remember" Finished Stages

Your Worker needs a memory of which stages are done so it can immediately close the input for any late-starting tasks (like restarts).

1. Add State to Worker Struct In `main.go`, add a map to track completed stages.

Go

```go
type Worker struct {
    // ... existing fields ...
    finishedStages map[int]bool // Add this
}
```

2. Initialize the Map In `main()`, inside the loop where you create the worker:

Go

```go
worker := Worker{
    // ... other fields ...
```

```
    finishedStages: make(map[int]bool), // Initialize it
}
```

3. Update `ReceiveFinishedStage` to Record Completion When the Leader tells you a stage is done, save that fact.

Go

```
func (w *Worker) ReceiveFinishedStage(stage int, reply *int) error {
    w.tasksLocker.Lock()
    w.finishedStages[stage] = true // <--- REMEMBER THIS

    // ... existing logic ...
    var inputsToClose []io.WriteCloser
    for key, value := range w.tasks {
        if key.Stage == stage+1 {
            inputsToClose = append(inputsToClose, value.input)
        }
    }
    w.tasksLocker.Unlock()

    // ... existing close loop ...
    println("Received stage ", stage, " completed")
    // ...
    return nil
}
```

4. Update `AddTask` to Check History This is the critical fix. When starting a task, check if its input source (the previous stage) is already finished.

Go

```
// Inside AddTask, right before returning:

    // ... (after your recovery logic is done) ...

    // NEW BLOCK: Check if we missed the "Finished" signal
    w.tasksLocker.RLock()
    prevStageFinished := w.finishedStages[t.Stage-1]
    w.tasksLocker.RUnlock()

    if prevStageFinished {
        // The previous stage is already done.
        // We must close this new task's input immediately so it
        // processes its recovered data (if any) and then finishes.
        _ = taskStdin.Close()
    }
```

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

Why this fixes it

With this change, when Task 8 restarts:

1. `AddTask` runs.
2. It sees `finishedStages[0]` is `true`.
3. It immediately closes `taskStdin`.
4. The `scanner` in the new task reads EOF.
5. The task finishes successfully and reports completion to the Leader.

User

Ok but, you can see that from the output of the worker that there are no outputs that any task in stage 0 completed and that stage 0 fully completed from the leader.
here is the output of the worker again:
Started Rainstorm task received new IPSreceived new IPSReceived kill task Killing task: &{0xc000203080 0xc000136778 0xc000136780 106 {13998767762373922721 29487703749 0x9cfc00} {13998767771561901715 38012006317 0x9cfc00} 97 0xc0001367b0} Scanner error: read |0: file already closed Task {1 8} failed: signal: killed Received stage -1 completed

Gemini

You are absolutely correct, and I apologize for the confusion in my previous explanation.

The log line `Received stage -1 completed` indicates that the **Source (File Generator)** has finished sending data to Stage 0. It does **not** mean Stage 0 has finished processing that data.

This distinction explains exactly why your worker is silent:

1. **Stage -1 Finished:** The inputs to Stage 0 were closed.
2. **Stage 0 Running:** Stage 0 tasks (likely on other workers, since they aren't in this log) are currently crunching through that data.
3. **Stage 1 Waiting:** Your restarted Task 8 (Stage 1) is **correctly waiting** for input from Stage 0. It hasn't received the `Received stage 0 completed` signal yet, so it keeps its input open and waits.

It is not sending completions because it is not done. It is effectively "blocked" waiting for Stage 0 to finish, which is the correct behavior.

However, you still have a Critical Race Condition

Even though the task is "correctly waiting" right now, you are walking into a trap.

If the Leader sends `Received stage 0 completed` **while Task 8 is restarting** (in the split second between `KillTask` and the new `AddTask`), your worker will miss the signal.

1. Task 8 dies.
2. **Leader sends `FinishedStage(0)`.**
3. Worker sees Task 8 is missing, so it does nothing.
4. Task 8 restarts (`AddTask`).
5. Task 8 opens its input and waits forever.

The Required Fix

To prevent this future deadlock, you **must** implement the `finishedStages` map we discussed. This ensures that even if the "Finished" signal arrives while the task is dead, the worker remembers it and immediately notifies the new task upon restart.

Here is the code again for clarity:

1. Add State to Worker

Go

```
type Worker struct {
    // ...
```

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

2. Initialize in `main`

Go

```
worker := Worker{
    // ...
    finishedStages: make(map[int]bool),
}
```

3. Update `ReceiveFinishedStage`

Go

```
func (w *Worker) ReceiveFinishedStage(stage int, reply *int) error {
    w.tasksLocker.Lock() // Change RLock to Lock
    w.finishedStages[stage] = true // Record the completion

    // ... calculate inputsToClose ...

    w.tasksLocker.Unlock() // Change RUnlock to Unlock

    // ... close inputs ...
    return nil
}
```

4. Update `AddTask`

Go

```
// Inside AddTask, right before returning:

    // Check if the previous stage is ALREADY done
    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
}
```

User

See this output, its getting stuck:
Started Rainstorm task received new IPS Received kill task Killing task: &{0xc0000d0a80 0xc00009c6d0 0xc00009c6d8 146 {13998768608835888741 28236156721 0x9cfc00} {13998768622568407476 41010031744 0x9cfc00} 141 0xc00009c708} Scanner error: read |0: file already closed Task {1 6} failed: signal: killed received new IPS Received stage -1 completed

Gemini

Based on the logs and the provided `main.go`, the issue is a **deadlock caused by holding a lock during blocking I/O**.

Here is the exact chain of events causing your job to get stuck:

1. **Stage 1 Task 6** restarts. It is likely processing recovered data or is simply slow, causing its **input pipe (stdin) to fill up**.
2. **Stage 0** is trying to send new tuples to Task 6.
3. Your **Tuple Listener** receives a tuple for Task 6.
4. Inside the listener (Line 284), you acquire `worker.tasksLocker.Lock()`.
5. **THE BUG:** You call `io.WriteString(t.input, split[3])` **while holding the lock**.
6. Because the pipe is full (Step 1), `io.WriteString` **blocks** indefinitely.
7. **The Deadlock:** Because the lock is held, **no other operation can proceed**.
   - The Leader tries to tell you `ReceivedFinishedStage(0)`, but that function needs the lock. It hangs.
   - Since `ReceivedFinishedStage(0)` never completes, the input to Stage 1 never closes.
   - Task 6 sits waiting for more input forever.

The Fix

You must **release the lock** before writing to the pipe.

Modify your `tupleListener` logic (around Line 280) to look like this:

Go

```
// ... inside the tupleListener loop ...

// 1. Acquire Lock ONLY to find the writer
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
                }

        // ... Send ACK logic ...
```

Why this fixes it

By moving `io.WriteString` outside the critical section, even if the write blocks (because the task is slow), the `tasksLocker` remains available. This allows `ReceiveFinishedStage` to acquire the lock, record that Stage 0 is done, and close the inputs, freeing the stuck tasks.

User

Ok I made those changes, but its still stuck, can I use a profiler or something to see where the code is getting stuck

Gemini

Yes, using a profiler is the best next step. Since your program is getting "stuck," you specifically want to look at the **goroutine stack traces** to see exactly which line every thread is pausing on.

Here are two ways to do this: the **Instant Way** (zero setup) and the **Robust Way** (using `pprof`).

Method 1: The Instant Way (SIGQUIT)

You can force Go to dump the stack trace of every running goroutine to your terminal without changing any code.

1. Run your worker as usual.
2. Wait for it to get stuck.
3. Press **`Ctrl`** + **`\`** (Backslash) in the terminal running the worker.

This sends a `SIGQUIT` signal. The program will exit and print a massive log of what every single goroutine was doing at that exact moment.

- **Look for:** `goroutine [running]` or `goroutine [semacquire]` (waiting for a lock).
- **Search for:** Your function names (e.g., `tupleListener`, `ReceiveFinishedStage`). See if they are stuck on a `.Lock()` call.

Method 2: The Robust Way (net/http/pprof)

If the output from Method 1 is too messy, use the `pprof` tool. It gives you an interactive shell to explore the stuck threads.

1. Modify `main.go`

Add the imports and start a debug server in a background goroutine at the very top of `main()`.

Go

```go
package main

import (
    // ... existing imports ...
    "net/http"
    _ "net/http/pprof" // Import for side effects
)

// ...

func main() {
    // 1. Start the Profiling Server
    go func() {
```

```
        fmt.Println("Pprof server running on localhost:6060")
        fmt.Println(http.ListenAndServe("localhost:6060", nil))
    }()

    // ... existing code ...
    leader, err := rpc.Dial("tcp", "fa25-cs425-1401.cs.illinois.edu"+IntroducePort)
    // ...
}
```

2. Run and Reproduce

Run your worker. Wait until it gets stuck.

3. Analyze with the Go Tool

Open a separate terminal window and run:

Bash

```
go tool pprof http://localhost:6060/debug/pprof/goroutine
```

This opens an interactive shell. Type the following commands:

- `top` : Shows which functions are currently active. If you see `sync.(*RWMutex).Lock` or `runtime.semacquire` at the top, you definitely have a deadlock.
- `traces` : Prints the stack trace of all goroutines. This is similar to Method 1 but cleaner.
- `list tupleListener` : Shows the source code of your listener and highlights the exact line where it is stuck.

What to search for in the output

You are looking for two goroutines fighting over the same lock.

Scenario A (Classic Deadlock):

- **Goroutine 1 (`tupleListener`):** Holding `tasksLocker` (or waiting for it) while trying to write to a pipe (`io.WriteString`).
- **Goroutine 2 (`ReceiveFinishedStage`):** Holding `tasksLocker` (or waiting for it) while trying to close pipes.

Scenario B (Channel Deadlock):

- **Goroutine 1:** Sending to `worker.taskOutputs` .
- **Goroutine 2:** The consumer of `worker.taskOutputs` is stuck (e.g., waiting on a network ACK), so the channel buffer fills up and blocks the sender.

If you paste the output of the stack trace (Method 1 or `traces` from Method 2), I can pinpoint exactly where the deadlock is.

User

Im using the goland ide from jet brains, is there a way to do this from the ide directly?

Gemini

Yes, GoLand has excellent built-in tools for this. You do not need to add any `pprof` code or extra imports.

Since your program is getting "stuck" (deadlock or waiting on a lock), the **Debugger's Goroutine View** is the fastest way to solve this.

Method 1: The Debugger (Best for "Stuck" Programs)

This method captures the exact state of every thread while the program is hanging.

1. **Run in Debug Mode:**
   - Click the **Green Bug Icon** (Debug) instead of the Run "Play" icon to start your worker.
   - Wait for the program to reach the "stuck" state.
2. **Pause the Program:**
   - In the **Debug Tool Window** (usually at the bottom), click the **Pause Program** button (two vertical bars `||` on the left side toolbar).
3. **Open the Goroutines Tab:**
   - Look for the **"Goroutines"** tab inside the Debugger window (it might be next to "Variables" and "Frames").
   - *If you don't see it:* Click the "Restore Layout" button or check the "Overhead" tabs on the right of the Debugger pane.
4. **Find the Blocked Goroutine:**
   - You will see a list of all active goroutines.
   - Look for groups that are in the `Waiting` state.
   - Expand the groups. Look specifically for your functions:
     - `main.func2` (This is your tuple listener)
     - `main.Worker.ReceiveFinishedStage`
   - Click on one. The **Frames** panel on the right will show you the exact line of code it is stuck on.

What to look for: If you see two different goroutines both stuck on a line calling `tasksLocker.Lock()` or `tasksLocker.RLock()`, you have found your deadlock.

Method 2: The Blocking Profiler (Visual Graph)

If the Debugger doesn't make it obvious, you can run the Blocking Profiler to see a graph of how long threads wait on resources.

1. **Right-Click your Main function:**
   - In the editor gutter (next to `func main`), right-click the Green Arrow.
   - Select **Profile 'go build main.go' with 'Blocking Profiler'**.
   - *(Note: Do not choose CPU Profiler; that is for high CPU usage. You want Blocking Profiler for stuck programs).*
2. **Reproduce the Stuck State:**
   - Let the program run until it gets stuck.

- Leave it stuck for about 10–20 seconds (so the profiler collects data on the wait).

3. **Stop the Profiler:**
   - Click the **Stop** button (Red Square) in the Run window.
4. **Analyze the Flame Graph:**
   - GoLand will open a "Flame Graph" tab.
   - Look for wide bars. These represent the time spent waiting.
   - If you see a large bar for `sync.(*RWMutex).Lock`, hover over it to see which function called it.

Recommendation

Use **Method 1 (Debug & Pause)** first. It allows you to inspect the values of variables (like `worker.tasks`) while the program is stuck, which helps verify if `finishedStages` is true or false.

Would you like me to analyze a screenshot of your Goroutines tab if you find the stuck threads?

User

Is this goroutine snapshot helpful for you?
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/RainStormWorker/main.go:459) 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 3 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 4 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 13 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 14 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) main.main.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:128) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.main (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:128)
Goroutine 15 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.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:227)  runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:132)
Goroutine 16  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.func2 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:254)  main.main.gowrap2 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:351)  runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:252)
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:323)
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:658)  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.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 25  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 27  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.
```

```
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
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).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.func2.1 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:262)  main.main.func2.gowrap1 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:349)  runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.main.func2 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:258)
Goroutine 31  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)  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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 34  runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)  runtime.selectgo
(/usr/lib/golang/src/runtime/select.go:351)  main.main.func3 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:361)  runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:355)
Goroutine 35  runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461)  runtime.selectgo
(/usr/lib/golang/src/runtime/select.go:351)  main.main.func4 (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:415)  runtime.goexit
(/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.main (/home/manandp2/g14-
mp4/RainStorm/RainStormWorker/main.go:408)
Goroutine 36  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)  net/rpc.(*Server).Accept.gowrap1
(/usr/lib/golang/src/net/rpc/server.go:636)  runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)
created at: net/rpc.(*Server).Accept (/usr/lib/golang/src/net/rpc/server.go:636)
Goroutine 42  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 44  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 46  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 48  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 50  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 52  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 54  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.
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 56  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.
```

```
(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:582)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 58  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)  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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 60  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)  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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 62  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)  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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 64  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)  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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581)  main.
(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614)
runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)  created at: main.(*Worker).AddTask
(/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 68  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) 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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581) main.(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.(*Worker).AddTask (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 70 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) 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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581) main.(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.(*Worker).AddTask (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 72 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) 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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581) main.(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.(*Worker).AddTask (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 74 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) 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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581) main.(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.(*Worker).AddTask (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 76 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) 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.(*Worker).AddTask.func1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:581) main.(*Worker).AddTask.gowrap1 (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:614) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693) created at: main.(*Worker).AddTask (/home/manandp2/g14-mp4/RainStorm/RainStormWorker/main.go:578)
Goroutine 79 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.gcBgMarkWorker

(/usr/lib/golang/src/runtime/mgc.go:1463) runtime.gcBgMarkStartWorkers.gowrap1
(/usr/lib/golang/src/runtime/mgc.go:1373) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)
created at: runtime.gcBgMarkStartWorkers (/usr/lib/golang/src/runtime/mgc.go:1373)
Goroutine 80 runtime.gopark (/usr/lib/golang/src/runtime/proc.go:461) runtime.gcBgMarkWorker
(/usr/lib/golang/src/runtime/mgc.go:1463) runtime.gcBgMarkStartWorkers.gowrap1
(/usr/lib/golang/src/runtime/mgc.go:1373) runtime.goexit (/usr/lib/golang/src/runtime/asm_amd64.s:1693)
created at: runtime.gcBgMarkStartWorkers (/usr/lib/golang/src/runtime/mgc.go:1373)

Gemini

Yes, this snapshot is extremely helpful. I have identified the exact cause of your stuck worker.

It is a **Circular Channel Deadlock** caused by your "Retry" logic.

The Diagnosis

1. **The Bottleneck:** The channel `worker.taskOutputs` (buffer size 100) is **full**.
2. **The Producers (Blocked):**
   - Look at **Goroutines 25, 27, 42, 44, 46, 48, 50, 52, 54, 56**.
   - They are all stuck at `main.go:582` (`c <- taskOutput`).
   - These are your tasks trying to send output. They are blocked because the channel is full.
3. **The Consumer (Blocked):**
   - Look at **Goroutine 15** (`main.main.func1`). This is the goroutine responsible for emptying the channel.
   - It is stuck at `main.go:227` : `worker.taskOutputs <- out` .
   - **The Bug:** The consumer pulled an item out, failed to process it (specifically, trying to send to Leader when `tupleSendConn` is nil), and tried to **push it back into the same channel**.

Why it deadlocks: The channel is full. The consumer holds an item. It tries to push it back. It cannot push because the channel is full. It waits for space. Space will never open up because the consumer (itself) is paused waiting to push.

The Fix

You must never perform a blocking write to `worker.taskOutputs` *inside* the goroutine that reads from `worker.taskOutputs` . You should perform the retry asynchronously.

Update your **Tuple Sender Goroutine** (around line 133 in `main.go`) to use a goroutine for retries.

Apply this pattern to ALL 5 places where you do `worker.taskOutputs <- out` inside that loop:

Go

```
// ... inside the consumer loop ...

// Example Fix for Line 227 (The specific one stuck in your logs)
if worker.tupleSendConn == nil {
    fmt.Println("CRITICAL ERROR: tupleSendConn is nil! Initialize hasn't run yet.")

    // BAD: worker.taskOutputs <- out
```

```
    // GOOD: Retry asynchronously to prevent blocking the consumer
    go func(t taskOutput) {
        time.Sleep(100 * time.Millisecond) // Wait a bit
        worker.taskOutputs <- t
    }(out)

    continue
}

// ...
// Apply the same fix for the Dial Error (Line 188), Write Error (Line 219), etc.
// ...
```

Why `tupleSendConn` is nil

The immediate trigger for the deadlock at Line 227 is that `worker.tupleSendConn` is `nil`. This implies that your tasks (Stage 2/Final Stage) are generating output **before** the `Initialize` RPC has successfully completed or connected to the Leader.

The async fix above will solve the deadlock (the worker will keep spinning and retrying until `Initialize` runs), but you should also ensure your Leader calls `Initialize` on the worker *before* it calls `AddTask`.

User

Ok so now this mostly works, but it duplicates some tuples

Gemini

Based on your `main.go`, the duplicates are caused by **Protocol Desynchronization** in your Tuple Sender logic.

When you timeout waiting for an ACK, you place the tuple back in the queue (`worker.taskOutputs <- t`) but you **do not close the connection**. This leaves the "Late ACK" (from the timed-out request) sitting in the TCP buffer. When you send the *next* tuple, you immediately read this old, stale ACK, leading to a mismatch, another retry, and a permanent loop of unnecessary resends and duplicate handling.

Here is the fix: **You must close the connection if you timeout or receive a bad ACK.**

Fix: Tuple Sender Loop (around line 238)

Find the block where you check the ACK and modify it to close the connection on failure.

Go

```
// ... inside the tuple sender loop ...

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

// FIX STARTS HERE
if err != nil || strings.TrimSpace(ack) !=
        fmt.Println("Ack failed or timed ou

        // 1. Close the connection to flush
        _ = client.Conn.Close()

        // 2. Remove from map so we dial fr
        worker.connectionsLock.Lock()
        delete(worker.connections, nextWork
        worker.connectionsLock.Unlock()

        // 3. Retry the tuple asynchronousl
        go func(t taskOutput) {
                time.Sleep(100 * time.Milli
                worker.taskOutputs <- t
        }(out)
}
```

Why this fixes it

By closing the connection, you ensure that the next attempt starts with a clean TCP stream. This prevents "Ghost ACKs" from the previous attempts from confusing the sender, which was causing your worker to aggressively re-send tuples that had actually been processed (overwhelming the receiver's de-duplication logic).

User

Ok, wait but now I just get a bunch of these: Started Rainstorm task received new IPS Received kill task Killing task: &{0xc000174d80 0xc0001367b0 0xc0001367b8 102 {13998775552682705692 2463523332 0x9d1c00} {13998775562892968195 12010109414 0x9d1c00} 100 0xc0001367e8} Scanner error: read |0: file already closed received new IPS Task {1 7} failed: signal: killed Ack failed or timed out, resetting connection: read tcp 172.22.94.231:36350->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:60932->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:60942->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:48988->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:48998->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:49000->172.22.94.231:8022: i/o timeout Received stage -1 completed Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33740->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33750->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33766->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:34598->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:34612->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:34620->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:34634->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:46176->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:46192->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:46194->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:58948->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:58954->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:58968->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:42876->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:42878->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:42882->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:42894->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:39460->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:39464->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:39466->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:45870->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:45876->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:45884->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33580->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33588->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33594->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:33604->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:59828->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:59832->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:59842->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:57464->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:57478->172.22.94.231:8022: i/o timeout Ack failed or timed out, resetting connection: read tcp 172.22.94.231:57480->172.22.94.231:8022: i/o timeout

Gemini

These logs confirm that your system is working logically, but it is suffering from **Network/Processing Congestion**.

The error `i/o timeout` means the Sender successfully sent the tuple, but the Receiver was too slow to process it and send back the "ACK" within your 3-second limit. This forces the Sender to close the connection and retry, which creates *more* overhead (TCP Handshakes) and makes the problem worse.

This is expected when processing large datasets with a "Stop-and-Wait" protocol (Send 1 tuple -> Wait for 1 ACK).

1. Immediate Fix: Increase the Timeout

Your `clientTimeout` of 3 seconds is too aggressive for a system under load (especially if the receiving task, like a Python script, is slow to read from Stdin).

In `main.go`, change line 68:

Go

```
// Change 3 seconds to 10 or 20 seconds
const clientTimeout = time.Second * 10
```

2. Secondary Fix: Optimize the Retry Sleep

Currently, you sleep for `100ms` on retry. If the receiver is overloaded, hammering it every 100ms won't help. Use a slightly longer backoff or a random jitter.

In your Tuple Sender loop (Lines 248-251):

Go

```
                                        go func(t taskOutput) {
                    // Increase wait time to let the receiver catch up
                                                time.Sleep(500 * time.Milli
                                                worker.taskOutputs <- t
                                        }(out)
```

Summary of what is happening in your logs:

1. **`Received kill task` / `Task failed`**: This is normal (assuming you triggered a failure).
2. **`Ack failed... i/o timeout`**: The worker is trying to send data to the next stage. The next stage is alive but busy.
3. **`resetting connection`**: Your fix is working! You are correctly closing the bad connection.
4. **Retries**: The system keeps trying until the Receiver catches up.

Once you increase the timeout, these errors should disappear or become very rare.

User

ok but shouldn't the deduplication map prevent the extra entries anyway?

Gemini

You are absolutely correct. **Logic dictates that your deduplication map should prevent duplicates** between workers, even if the sender retries 100 times.

If you are seeing actual duplicate data records in your output, it is because of a specific blind spot in your code: **The connection to the Leader.**

1. The Blind Spot: Final Stage Output

Your deduplication logic exists **only on the Worker**. The Leader (which writes the final output to HyDFS) has no such logic because you strip the Tuple ID before sending data to it.

Look at `main.go` lines 255-260 (in the `else` block for the final stage):

Go

```
// Output data to the distributed file system
// Send the tuple to the leader...
_, err = fmt.Fprintln(worker.tupleSendConn, out.output) // <--- PROBLEM
```

You are sending **only** the content (`out.output`). You are **not** sending the `out.taskId` or `out.tupleId` .

- **Scenario:** The Worker sends a final result to the Leader.
- **Network:** The Leader receives it and writes it to HyDFS.
- **Timeout:** The Worker doesn't get an TCP ACK from the Leader fast enough (network congestion).
- **Retry:** The Worker's retry loop triggers (`go func ... worker.taskOutputs <- t`).
- **Duplicate:** The Worker sends the result *again*.
- **Leader:** Since the Leader just sees a stream of strings (no IDs), it assumes this is a new result and writes it to HyDFS again.

To fix this: You would need to send the ID to the Leader and have the Leader perform deduplication, just like your Workers do.

---

2. Why the "Ack Failed" logs are okay

The logs you posted: `Ack failed or timed out, resetting connection: ... i/o timeout`

These logs are **not** proof of duplicates. They are actually proof that your system is working correctly under heavy load (congestion).

1. **Sender:** Sends Tuple A.
2. **Receiver:** Gets Tuple A, processes it, writes to dedupe map.
3. **Receiver:** Tries to send "ACK", but the network is clogged.
4. **Sender:** Times out. Resets connection. Retries Tuple A.
5. **Receiver:** Gets Tuple A again. **Checks Map -> Exists.** Rejects it. Sends ACK.

So, for Worker-to-Worker communication, your map **is** protecting you. The log spam is annoying, but the data is safe.

3. Summary of the Real Issue

- **Worker-to-Worker:** Safe. Your `receivedTuples` map handles the retries seen in the logs.
- **Worker-to-Leader: Unsafe.** Because you strip the IDs, any retry on the final stage results in permanent duplicate data in your output file.

Recommendation: For the purposes of this assignment, if you cannot change the Leader code, you just have to accept that `i/o timeouts` on the final stage will cause duplicates. Minimizing the timeouts (by increasing `clientTimeout` to `10s` or `20s` as suggested) is the best mitigation.

User

No, I can change the leader, I wrote it myself, can you show me where to add it. Here is the leader?

Gemini

This is great! Since you own the Leader code, we can implement **Deduplication at the Source (Leader)**. This is the "Silver Bullet" for your duplicate problem because it guarantees that even if a Worker sends the same result 10 times, the Leader writes it to the file only once.

Here is the plan:

1. **Protocol Update:** We will change the format sent to the Leader from just `Content` to `TaskID-TupleID,Content`.
2. **Leader Update:** The Leader will track processed IDs. If it sees a duplicate ID, it ignores the write but sends back an ACK (so the Worker stops retrying).

Step 1: Update `RainStorm` Struct

Add a map to track the final tuples you have already written to disk.

Go

```go
type RainStorm struct {
    // ... existing fields ...

    // NEW: Deduplication State
    FinalTuplesLock sync.RWMutex
    FinalTuples     map[string]bool
}
```

Step 2: Initialize the Map

In `main()`, right after `r.LogFileChan = make...`:

Go

```go
                r.LogFileChan = make(chan string, 100)
                r.StartTime = time.Now()

        // NEW: Initialize Map
        r.FinalTuples = make(map[string]bool)
```

Step 3: Rewrite the Tuple Listener (Producer 1)

This is the core logic change. Replace the **entire** `TupleListener` block (lines 236–275) with this logic.

Key Changes:

1. It parses the input string to separate the **ID** from the **Content**.
2. It checks `r.FinalTuples`.
3. It sends an **ACK** back to the Worker (stopping the Worker's retry loop).

Go

```
// --- TUPLE LISTENER (PRODUCER 1) ---
wgProducers.Add(1)
go func() {
        defer wgProducers.Done()

        go func() {
                <-ctx.Done()
                tupleListener.Close()
        }()

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

                wgProducers.Add(1)
                go func(c net.Conn) {
                        defer wgProducers.Done()
                        defer c.Close()

                        // Sidecar closer
                        go func() { <-ctx.Done(); c.Close() }()

                        reader := bufio.NewReader(c)
                        for {
                                // 1. Read the message
                                line, err := reader.ReadString('\n')
                                if err != nil {
                                        return
                                }

        line = strings.TrimSpace(line)
                                if len(line) == 0 {
                                        continue
                                }

                                // 2. Parse Protocol: "TaskID-TupleID,Conte
                                // We split by the first comma only
                                parts := strings.SplitN(line, ",", 2)
                                if len(parts) < 2 {
                                        fmt.Println("Invalid tuple format r
                                        continue
                                }
```

```
                                        id := parts[0]      // e.g., "1:4-50"
                                        content := parts[1] // e.g., "Count: 100"

                                        // 3. Deduplication Check
                                        r.FinalTuplesLock.Lock()
                                        seen := r.FinalTuples[id]
                                        if !seen {
                                                r.FinalTuples[id] = true // Mark as
                                        }
                                        r.FinalTuplesLock.Unlock()

                                        // 4. Write to File (Only if NEW)
                                        if !seen {
                                                localOutputFile.WriteString(content
                                                outputChan <- content + "\n"
                                        }

                                        // 5. Send ACK (ALWAYS send ACK, even if du
                                        // The worker is waiting for this to stop r
                                        // Format: "TaskID-TupleID-ACK"
                                        ackMsg := fmt.Sprintf("%s-ACK\n", id)
                                        _, _ = c.Write([]byte(ackMsg))
                                }
                        }(conn)
                }
        }()
```

Step 4: Update the Worker (Crucial!)

You must update your **Worker code** to send the data in this new format (`ID,Content`) and wait for the ACK.

In your `Worker` (lines 255-260 of previous code):

Go

```
// ... inside tuple sender loop ...

if isFinalStage {
    // OLD: _, err = fmt.Fprintln(worker.tupleSendConn, out.output)

    // NEW: Send ID + Content
    msg := fmt.Sprintf("%s-%d,%s\n", out.taskId.String(), out.tupleId, out.output)
    _, err = worker.tupleSendConn.Write([]byte(msg))

    // NEW: Wait for Leader ACK (Matches your normal retry logic)
    // You can copy/paste your existing ACK check logic here,
```

```
    // or wrap the Leader connection in a struct that behaves like a worker client.
}
```

Why this fixes duplicates:

1. **Worker:** Sends "Tuple 50".
2. **Leader:** Writes "Tuple 50". Tries to send ACK. Network drops ACK.
3. **Worker:** Times out. Resends "Tuple 50".
4. **Leader:** Sees "Tuple 50" in map. **Does NOT write to file.** Sends ACK again.
5. **Worker:** Gets ACK. Success.

Result: File has exactly one copy of "Tuple 50".

User

Ok, ignore this for now, it's getting stuck again when a failure occurs. This is what I have right now

Gemini

Based on the provided `main.go` and the logs, the "stuck" behavior occurring after a failure (or kill) is primarily due to **reporting "Failure" for tasks that were intentionally Killed.**

When the Leader kills a task (e.g., during rebalancing or shutdown), your worker currently reports it as a `ReceiveFailure`. This confuses the Leader, causing it to potentially restart a task that it intended to stop, leading to duplicate tasks, race conditions, or "zombie" tasks fighting for resources.

Here is the solution:

1. Fix: Stop Reporting "Failure" on Intentional Kills

In `AddTask`, you monitor the process output and wait for it to exit. When you kill a task via `KillTask`, you remove it from the `w.tasks` map. You must check this map **before** sending the `ReceiveFailure` RPC.

Modify the goroutine in `AddTask` (around lines 480-510):

Go

```
        // ... inside AddTask ...
        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++
                }

                // It is normal to get an error here if we closed the pipe during KillTask
                if scanner.Err() != nil {
                        fmt.Println("Scanner stopped (task ending):", scanner.Err())
                }

                err = cmd.Wait()

                // CRITICAL FIX: Check if task was intentionally removed (Killed)
                w.tasksLocker.RLock()
                _, stillActive := w.tasks[tId]
                w.tasksLocker.RUnlock()

                if err == nil {
                        // SUCCESS CASE
```

```
                        var reply int
                        println("told leader im done", t.String())
                        // Only report completion if it's still active (optional, but safer
                        if stillActive {
                                err = w.rainStormLeader.Call("RainStorm.ReceiveTaskCompleti
                                if err != nil {
                                        println(err.Error())
                                }
                                println("leader responded", t.String())
                        }
                } else {
                        // FAILURE CASE
                        // ONLY report failure if the task is supposed to be running!
                        if stillActive {
                                var reply int
                                fmt.Printf("Task %v failed unexpectedly: %v. Reporting to L
                                _ = w.rainStormLeader.Call("RainStorm.ReceiveFailure", t, &
                        } else {
                                fmt.Printf("Task %v killed intentionally. No failure report
                        }
                }

                // Cleanup (Only if it wasn't already deleted by KillTask)
                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)
```

2. Verify: Finished Stage Logic for Restarts

You have correctly implemented the `finishedStages` logic (Line 551), checking `t.Stage-1`.

- Ensure that your Leader sends `ReceiveFinishedStage(-1)` when the input generation is done.
- Your log `Received stage -1 completed` confirms this is working.
- **Result:** If a Stage 0 task crashes and restarts, `isPrevStageDone` will be true (because `finishedStages[-1]` is true). The input closes, and the task finishes gracefully. This part of your code is correct.

3. Tuning: Reduce Network Backoff

In your Tuple Sender loop (Line 237, 248, 260), you are sleeping for `100ms` on retry. If you have a high volume of tuples, this is too slow and causes the `taskOutputs` channel to fill up, blocking your tasks.

Recommendation: Use a randomized, shorter backoff for network retries, or simply yield.

Go

```
// Replace:
// time.Sleep(100 * time.Millisecond)

// With:
time.Sleep(10 * time.Millisecond) // Faster retry loop
```

4. Summary of the "Stuck" State

The "stuck" state was likely a logical deadlock:

1. Leader calls `KillTask`.
2. Worker kills process -> `cmd.Wait()` returns error.
3. Worker reports `ReceiveFailure` (Old Bug).
4. Leader sees failure -> Restarts Task (`AddTask`).
5. **Simultaneously**, Leader might have moved the task elsewhere.
6. You now have a "Zombie" task running on this worker that the Leader doesn't account for correctly, or the Worker is stuck waiting for a connection to a next-stage worker that hasn't been updated yet.

By implementing Fix #1, you ensure that `KillTask` is final and does not trigger error recovery logic.

User

No, it's ok, I want the task to tell the leader it's failed because then it will restart it, which is what I want to happen. The restart is not necessarily on the same vm, but the "kill_task" on the leader doesn't tell the leader itself to do anything, it is only telling the worker to emulate a failing task

Gemini

You are absolutely right. If you are using `KillTask` to simulate failure ("Chaos Monkey" style), then the Worker **must** report the failure to the Leader so the Leader knows to reschedule it.

My previous advice to suppress the failure report was based on the assumption that `KillTask` was a graceful administrative stop. Since it is for testing fault tolerance, **you should keep the failure reporting logic exactly as you have it in your uploaded file.**

Based on your logs and code, your system is "stuck" because of **Network Congestion** caused by aggressive timeouts, and possibly invisible restarts.

Here are the 3 steps to fix the "stuck" state and verify restarts:

1. Fix the "Retry Storm" (Critical)

Your logs show `Ack failed ... i/o timeout`. This is happening because your timeout is **3 seconds** (`clientTimeout`) and your retry sleep is **100ms**. When a task restarts, the system is slow. The sender times out, closes the connection, and retries. This creates a loop of "Connect -> Timeout -> Close -> Connect" that floods the network and prevents actual data from getting through.

In `main.go`, change these constants:

Go

```go
// Increase timeout to 10s or 20s to survive system lag
const clientTimeout = time.Second * 10

// In your Tuple Sender goroutine (lines 237, 248, 260):
go func(t taskOutput) {
    // Increase sleep to 1s to give the network time to breathe
    time.Sleep(1 * time.Second)
    worker.taskOutputs <- t
}(out)
```

2. Verify the Restart (UX Fix)

Your `AddTask` function is currently "silent" on the console. It writes to a log file, but you can't see that in your terminal output. This makes it look like the Worker is stuck when it has actually restarted the task.

Add a print statement to `AddTask` (around line 430):

Go

```go
func (w *Worker) AddTask(t Task, reply *int) error {
    fmt.Printf("Starting (or Restarting) Task: %d-%d\n", t.Stage, t.TaskNumber) // <--- ADD

    // Set up the task and its pipes
```

```
    cmdArgs := strings.Fields(t.Executable.Args)
    // ...
```

3. Prevent Data Loss on Crash

There is a race condition in your `tupleListener`. If `KillTask` kills the task **after** the tuple is marked as "RECEIVED" in `worker.receivedTuples` but **before** the task processes it, that tuple is lost forever (because the new task will reject it as a duplicate).

To fix this, ensure the restart logic **reprocesses** the tuple log. Your code already attempts this (`recoveredTask` block), but relies on HyDFS. Ensure your `tupleListener` flushes logs quickly.

In `main.go`, Tuple Listener (Line 368): You send `RECEIVED` to `logChan`. This is good. Ensure your logger (Line 415) handles high volume by potentially decreasing the flush interval or buffer size slightly if you notice data loss.

Summary

1. **Do not change the failure reporting.** Your current code correctly tells the Leader about the failure.
2. **Increase `clientTimeout`** to `10s`.
3. **Increase retry sleep** to `1s`.
4. **Add a `fmt.Println`** in `AddTask` so you can verify the restart visually.

User

Ok wait these are the logs, I removed the closing connection remember? I also attached the code to look at again
Started Rainstorm task received new IPS Received kill task Killing task: (pid 10385) &{0xc0000afe00 0xc000048890 0xc000048898 107 {13998777813527670848 3366027104 0x9d0c00} {13998777822833288782 12007968622 0x9d0c00} 97 0xc0000488c8} Scanner error: read |0: file already closed received new IPS Task {1 8} failed: signal: killed Received stage -1 completed

Gemini

You have reintroduced the **TCP Desynchronization Bug** by removing `client.Conn.Close()`.

When you removed that line, you created the following trap:

1. **Timeout:** You send `Tuple A`. The network lags. You timeout waiting for `ACK A`.
2. **Retry:** You enter the `if err != nil` block. You queue `Tuple A` to send again. **But you keep the socket open.**
3. **Desync:** You send `Tuple A` (Attempt 2).
4. **Ghost Read:** Immediately, your reader picks up the **Late** `ACK A` (from Attempt 1) that just arrived.
5. **Mismatch/Confusion:** The code might accept this ACK (if IDs match) or reject it. If it accepts it, you are now one step behind in the TCP stream. The *next* read will get the ACK for Attempt 2, but you will be looking for the ACK for `Tuple B`.

This causes your worker to spin endlessly, flooding the network with retries and eventually filling up the `taskOutputs` channel, which causes the "stuck" behavior (deadlock) because the tasks can't write to the full channel.

The Fix

You **must** close the connection if the ACK fails or times out. This is the only way to "flush" the dirty TCP stream and force a clean handshake for the next attempt.

Here is the corrected block for your `main.go`.

1. Fix the Tuple Sender (Lines 251-255 in your file) Replace your current ACK check with this:

Go

```
                                // Wait for the ack
                                _ = client.Conn.SetReadDeadline(time.Now().
                                ack, err := client.Buf.ReadString('\n')
                                expectedAck := fmt.Sprintf("%s-%d-%s", out.

                                if err != nil || strings.TrimSpace(ack) !=
                        // LOGGING (Optional but helpful)
                        // fmt.Println("Ack failed, closing connection to reset:", err)

                        // 1. CRITICAL: Close the connection to flush the stream
                                        _ = client.Conn.Close()

                        // 2. Remove from map so we dial fresh next time
                                        worker.connectionsLock.Lock()
```

```
                                        delete(worker.connections, nextWork
                                        worker.connectionsLock.Unlock()

                        // 3. Retry asynchronously with a longer backoff
                                        go func(t taskOutput) {
                                                time.Sleep(1 * time.Second)
                                                worker.taskOutputs <- t
                                        }(out)
                                }
```

2. Increase Timeout (Line 73) Your 3-second timeout is too strict for a restarting system.

Go

```
// main.go Line 73
const clientTimeout = time.Second * 10 // Increase to 10s
```

Why the logs look "Stuck"

Your logs show: `Task {1 8} failed: signal: killed` `Received stage -1 completed`

This is actually correct behavior for a worker that is **waiting**.

1. **Task {1, 8}** (Stage 1) died and likely restarted (add the print statement to `AddTask` to confirm this!).
2. **Stage -1** (Generator) is done.
3. **Stage 0** is still running (sending data).
4. **Task {1, 8}** is waiting for data from Stage 0.

If it seems stuck, it is because the Stage 0 workers are struggling to connect to your restarted task (likely because of the `taskOutputs` channel deadlock described above). Fixing the `Close()` logic will free up the Stage 0 workers to successfully retry and reconnect to your new task.

User

Ok here is a rerun, it's getting stuck/going really slow after the failure. I also included go routine dumps after it slowed/stalled. Here is the output:
Started Rainstorm task Starting task {0 0 {Filter }} with pid: 11187 Starting task {1 0 {Filter }} with pid: 11192 Starting task {2 0 {Filter }} with pid: 11196 Starting task {3 0 {Filter }} with pid: 11202 Starting task {4 0 {Filter }} with pid: 11207 Starting task {5 0 {Filter }} with pid: 11212 Starting task {6 0 {Filter }} with pid: 11217 Starting task {7 0 {Filter }} with pid: 11222 Starting task {8 0 {Filter }} with pid: 11227 Starting task {9 0 {Filter }} with pid: 11232 Starting task {0 1 {Transform }} with pid: 11237 Starting task {1 1 {Transform }} with pid: 11238 Starting task {2 1 {Transform }} with pid: 11246 Starting task {3 1 {Transform }} with pid: 11250 Starting task {4 1 {Transform }} with pid: 11255 Starting task {5 1 {Transform }} with pid: 11256 Starting task {6 1 {Transform }} with pid: 11264 Starting task {7 1 {Transform }} with pid: 11268 Starting task {8 1 {Transform }} with pid: 11273 Starting task {9 1 {Transform }} with pid: 11277 received new IPS Received kill task Killing task: (pid 11246) &{0xc00012e780 0xc00004a7b0 0xc00004a7b8 104 {13998778522587448365 5560642927 0xbdd380} {13998778531636554965 14019814950 0xbdd380} 99 0xc00004a7e8} Scanner error: read |0: file already closed Starting task {0 1 {Transform }} with pid: 11284 received new IPS Task {1 2} failed: signal: killed Received stage -1 completed

Gemini

Based on the `goroutines.txt` dump and the `main.go` file you provided, I can confirm **exactly** why your system is stalling.

The diagnosis is the same as before: **You are missing the Connection Reset fix.** The stack traces prove that your Worker is deadlocked because the "Consumer" cannot clear the channel fast enough, causing all "Producers" (your tasks) to block.

The Evidence from `goroutines.txt`

1. **The Producers are Blocked (Your Tasks):**
   - Goroutines **29, 31, 33, 38, 40, 46...** (and many others) are all stuck at **Line 599**: `c <- taskOutput{...}`.
   - This means your `worker.taskOutputs` channel (buffer size 100) is **full**. Your tasks are trying to output data, but there is no room, so they pause execution.
2. **The Consumer is Struggling (Tuple Sender):**
   - Goroutine **11** is your Tuple Sender. It is currently at **Line 230**: `ack, err := client.Buf.ReadString('\n')`.
   - It is waiting for an ACK. When it times out (after 3 seconds), your current code (Lines 231-235) simply **re-queues the tuple** (`worker.taskOutputs <- t`) and loops again.
   - **The Fatal Flaw:** You **do not close the connection**.
   - Because the connection is left open, the TCP stream stays "dirty." The next time you try to send a tuple on this connection, it fails immediately or reads garbage, triggering *another* retry.
   - This endless loop of "Send -> Fail -> Retry" is too slow to drain the channel. The channel fills up, and the whole worker stalls.

The Solution

You must apply the fix to **close the connection** when an ACK fails. This forces a fresh, clean connection for the next attempt.

Update `main.go` at Line 231 (inside the Tuple Sender loop):

Go

```
// ... inside the tuple sender loop ...

// Current Code (BROKEN):
// if err != nil || strings.TrimSpace(ack)
//      go func(t taskOutput) {
//              time.Sleep(100 * time.Milli
//              worker.taskOutputs <- t
//      }(out)
// }

// NEW CORRECT CODE (FIXED):
if err != nil || strings.TrimSpace(ack) !=
        // 1. Close the connection to clean
        _ = client.Conn.Close()

        // 2. Remove from map so we dial fr
        worker.connectionsLock.Lock()
        delete(worker.connections, nextWork
        worker.connectionsLock.Unlock()

        // 3. Retry asynchronously (Slower
        go func(t taskOutput) {
                time.Sleep(1 * time.Second)
                worker.taskOutputs <- t
        }(out)
}
```

Additional Tuning (Recommended)

To further prevent stalls during restarts, update these constants in `main.go`:

1. **Line 68:** Increase timeout to survive network blips.

   Go

   ```
   const clientTimeout = time.Second * 10 // Change 3 to 10
   ```

2. **Lines 195 & 244:** Increase retry backoff. Change `100 * time.Millisecond` to `1 * time.Second`. This gives the restarting worker time to actually come back online before you hammer it with retries.

Once you add the `client.Conn.Close()` and `delete(...)` lines, the stall will disappear.

User

Ok these are the new file, log, and output:
Started Rainstorm task Starting task {0 0 {Filter }} with pid: 12307 Starting task {1 0 {Filter }} with pid: 12313 Starting task {2 0 {Filter }} with pid: 12318 Starting task {3 0 {Filter }} with pid: 12323 Starting task {4 0 {Filter }} with pid: 12327 Starting task {5 0 {Filter }} with pid: 12332 Starting task {6 0 {Filter }} with pid: 12336 Starting task {7 0 {Filter }} with pid: 12341 Starting task {8 0 {Filter }} with pid: 12346 Starting task {9 0 {Filter }} with pid: 12352 Starting task {0 1 {Transform }} with pid: 12356 Starting task {1 1 {Transform }} with pid: 12361 Starting task {2 1 {Transform }} with pid: 12366 Starting task {3 1 {Transform }} with pid: 12371 Starting task {4 1 {Transform }} with pid: 12372 Starting task {5 1 {Transform }} with pid: 12379 Starting task {6 1 {Transform }} with pid: 12383 Starting task {7 1 {Transform }} with pid: 12388 Starting task {8 1 {Transform }} with pid: 12394 Starting task {9 1 {Transform }} with pid: 12399 received new IPS Received kill task Killing task: (pid 12366) &{0xc0001dea80 0xc00004a668 0xc00004a670 106 {13998778952457130350 3843249018 0xbdd380} {13998778962290153347 13012595600 0xbdd380} 104 0xc00004a6a0} Scanner error: read |0: file already closed Starting task {0 1 {Transform }} with pid: 12412 received new IPS Task {1 2} failed: signal: killed Received stage -1 completed

Gemini

Based on the `goroutines.txt` dump and the `main.go` file you uploaded, I have verified that your "stuck" state is actually a **Network Timeout Freeze**.

The Smoking Gun

In `goroutines.txt`, **Goroutine 9** is your Tuple Sender loop (`main.main.func1`). It is currently parked in a network wait state:

- **Status:** `poll.runtime_pollWait`
- **Location:** `main.go:230` (in the uploaded file).
- **Code:** `ack, err := client.Buf.ReadString('\n')`

This confirms that your worker is strictly serial: it sends **one tuple** and waits up to **3 seconds** (your `clientTimeout`) for an ACK. If the destination worker is slow, overloaded, or restarting, your sender sits idle for 3 seconds.

- **Result:** You process 1 tuple every 3 seconds.
- **Effect:** The upstream tasks (Goroutines 29-46) fill the `taskOutputs` channel (buffer 100) instantly and then block. The whole system appears "stuck" because it is crawling at a snail's pace.

The Fix: 3-Step Performance Booster

You need to make the Sender fail faster and buffer more data.

1. Use `DialTimeout` (Crucial Prevention of Hangs)

Your current `net.Dial` (Line 177) has no timeout. If a remote worker crashes and the IP becomes unreachable (e.g., packets dropped), `net.Dial` can hang for **minutes**. You must cap this.

In `main.go`, replace line 177:

Go

```
// OLD: conn, err := net.Dial("tcp", nextWorker+TuplePort)
// NEW: Use a timeout so we don't freeze the loop if a node is down
```

```
conn, err := net.DialTimeout("tcp", nextWorker+TuplePort, 2*time.Second)
```

2. Increase Channel Buffer

Your current buffer of `100` is too small for the "bursts" of traffic that happen after a restart. Increase it to keep tasks running while the network catches up.

In `main.go`, line 90:

Go

```
// OLD: taskOutputs: make(chan taskOutput, 100),
// NEW: Give it breathing room
taskOutputs: make(chan taskOutput, 5000),
```

3. Parallelize the Sender (Optional but Recommended)

Because your sender loop is serial, one bad connection blocks **all** other connections. The robust fix is to process the `taskOutputs` channel using a pool of goroutines, but a quick "patch" is to use a **short timeout** for the ACK so you cycle through retries faster.

In `main.go`, line 68:

Go

```
// Reduce the ACK wait time. If they don't reply in 2s, assume failure and retry.
const clientTimeout = time.Second * 2
```

Summary Check

Your uploaded `main.go` **DOES** have the `client.Conn.Close()` fix (Lines 252 & 220). **Do not remove it.** It is saving you from data corruption. The "stuck" behavior is purely due to the serial blocking nature of the loop combined with network delays.

Action Plan:

1. Change `net.Dial` to `net.DialTimeout`.
2. Increase `taskOutputs` buffer to `5000`.
3. Run it again. The "stuck" state should turn into "flowing" logs.

User

hmm, it is still stuck
Started Rainstorm task Starting task {0 0 {Filter }} with pid: 12696 Starting task {1 0 {Filter }} with pid: 12700 Starting task {2 0 {Filter }} with pid: 12705 Starting task {3 0 {Filter }} with pid: 12709 Starting task {4 0 {Filter }} with pid: 12714 Starting task {5 0 {Filter }} with pid: 12719 Starting task {6 0 {Filter }} with pid: 12724 Starting task {7 0 {Filter }} with pid: 12729 Starting task {8 0 {Filter }} with pid: 12734 Starting task {9 0 {Filter }} with pid: 12739 Starting task {0 1 {Transform }} with pid: 12744 Starting task {1 1 {Transform }} with pid: 12749 Starting task {2 1 {Transform }} with pid: 12754 Starting task {3 1 {Transform }} with pid: 12759 Starting task {4 1 {Transform }} with pid: 12764 Starting task {5 1 {Transform }} with pid: 12768 Starting task {6 1 {Transform }} with pid: 12770 Starting task {7 1 {Transform }} with pid: 12778 Starting task {8 1 {Transform }} with pid: 12783 Starting task {9 1 {Transform }} with pid: 12788 received new IPS Received kill task Killing task: (pid 12778) &{0xc0000c8c00 0xc000098620 0xc000098628 99 {13998779302620173742 22451325218 0xbdf380} {13998779311854249087 31021724124 0xbdf380} 93 0xc000098658} Scanner error: read |0: file already closed Starting task {0 1 {Transform }} with pid: 12795 received new IPS Task {1 7} failed: signal: killed

Gemini

Based on the detailed `goroutines.txt` dump and the provided source code, I have pinpointed the exact cause of the "stuck" state. It is not a deadlock in the traditional sense, but a **Throughput Collapse** caused by your serial networking protocol.

The Diagnosis

1. **The Bottleneck (Goroutine 12 - Tuple Sender):** Your system has **one** single goroutine responsible for emptying the `taskOutputs` channel.
   - It operates in a **Stop-and-Wait** loop: Send 1 Tuple -> Wait for ACK -> Send Next.
   - The trace shows it sitting at `client.Buf.ReadString('\n')`.
   - If the destination worker is slow, restarting, or the network is laggy, this waits for up to **1 second** (your timeout).
   - **Result:** Your entire cluster is limited to processing **1 tuple per second**.
2. **The Backpressure (Goroutines 24-72):** Because the sender is clearing the buffer at 1 item/sec, the `taskOutputs` channel (buffer 5000) fills up almost instantly.
   - **Goroutines 24, 26, 28...** (your `AddTask` outputs) are all blocked on `c <- taskOutput`.
   - This stops your tasks (Filter/Transform) from running.
3. **The "Lock" Illusion (Goroutines 73+):** You see hundreds of goroutines waiting on `worker.tasksLocker.Lock()`. This is a side effect.
   - Because the tasks are paused, the input pipes fill up.
   - The Tuple Listener (Goroutine 13/73) receives data from the network but blocks when trying to write to the filled pipe.
   - While blocked, it holds the connection open, causing the *upstream* worker to also stall (creating a chain reaction).

The Fix: Parallel Pipelining

You must parallelize the Sender to allow multiple ACKs to be in-flight simultaneously. Since your `WorkerClient` (connection) cannot be safely shared across threads, the simplest robust fix is to **launch multiple sender threads, each with its own connection cache.**

Modify `main.go` inside the `main()` function (around line 100):

Replace your single `go func() { ... }()` block for the sender with this **Worker Pool**:

Go

```
// --- PARALLEL TUPLE SENDERS ---
// Launch 16 sender threads to drain the channel faster
for i := 0; i < 16; i++ {
        wg.Add(1)
        go func(threadID int) {
                defer wg.Done()
                // Each thread gets its OWN connection cache to avoid locki
                // and protocol mixing.
                localConnections := make(map[string]*WorkerClient)

                for {
                        select {
                        case <-ctx.Done():
                                // Cleanup local connections
                                for _, client := range localConnections {
                                        client.Conn.Close()
                                }
                                return
                        case out := <-worker.taskOutputs:
                                // ... (COPY ALL YOUR EXISTING LOGIC HERE)

                                // IMPORTANT CHANGE 1: Use 'localConnection
                                // Remove all 'worker.connectionsLock' usag

                                // ...

                                // IMPORTANT CHANGE 2: Fix the Dial logic t
                                // nextWorker := ...
                                client, ok := localConnections[nextWorker]
                                if !ok {
                                        conn, err := net.DialTimeout("tcp",
                                        if err != nil {
                                                // Retry logic...
                                                continue
                                        }
                                        client = &WorkerClient{Conn: conn,
```

```
                                        localConnections[nextWorker] = clie
                                }

                                // ... (Rest of send logic) ...

                                // IMPORTANT CHANGE 3: Handle Errors by del
                                if err != nil || strings.TrimSpace(ack) !=
                                        _ = client.Conn.Close()
                                        delete(localConnections, nextWorker

                                        // Retry async...
                                }
                        }
                }
        }(i)
}
```

Why this fixes it

1. **Multiplied Throughput:** Even if one connection waits 1s for a timeout, the other 15 threads continue processing tuples. Your throughput jumps from 1 tuple/sec to ~16+ tuples/sec immediately.
2. **Fault Isolation:** A slow destination worker only blocks one thread, not the entire system.
3. **No Lock Contention:** By giving each thread its own map, you remove the `connectionsLock` bottleneck completely.