

## gemini-conversation

Exported on: 12/7/2025, 10:23:54 PM

---

**User**

Ok I want some help fixing my streaming worker getting stuck after a failure. I think the issue is a channel filling up or a lock being held for too long, but not sure. Here is the code and the goroutine dump after a failure and it getting stuck.

**Gemini**

Based on the code and the goroutine dump provided, the worker is entering a state of **channel starvation/livelock** caused by your retry logic.

Here is the breakdown of why the worker gets stuck and how to fix it.

## The Diagnosis: Channel Starvation

The root cause is a circular dependency on the `worker.taskOutputs` channel during failure scenarios.

### 1. The Channel is Full:

- Goroutine 15 (and many others like 35, 37, 39) are **Task Producers**. They are reading output from your executables and trying to send it to `worker.taskOutputs`.
- The dump shows them blocked at `runtime.chansend`. This means the channel buffer (100) is completely full.



### 2. The Retry Loop:

- Goroutine 78 (and 79, 80... 98) are **Retry Goroutines**.
- When your Consumer fails to send a tuple (e.g., `net.Dial` fails or no ACK), it spawns a new goroutine to sleep and then **push the tuple back into `worker.taskOutputs`**:

**Go**

```
go func(t taskOutput) {
    time.Sleep(100 * time.Millisecond) // Wait a bit
    worker.taskOutputs <- t
}(out)
```

- The dump shows these goroutines are *also* blocked at `runtime.chansend`.



### 3. The Deadlock Mechanism:

- When a downstream node fails, every attempt to send fails. The Consumer pulls a tuple, fails, and spawns a Retry Goroutine.
- While the Retry Goroutine sleeps (100ms), the **Task Producers** (which are much faster) immediately fill the empty slot in the `taskOutputs` channel.
- When the Retry Goroutine wakes up, it tries to write to `taskOutputs`, but the channel is full. It blocks.
- The Consumer pulls the *next* fresh tuple, fails, and creates *another* Retry Goroutine.
- **Result:** You accumulate hundreds of blocked Retry Goroutines (G78-G98) that can never re-enter the queue because the Task Producers (G15) beat them to the lock every time. The old data is stuck in memory, and the system churns through new data until everything locks up.

## The Fix: Separate the Retry Channel

To fix this, you must stop putting failed tuples back into the same channel that your fast producers are filling. You need a dedicated "priority" lane for retries.

### Step 1: Add a Retry Channel to your Worker struct

In `main.go`, add `retryChan` to the `Worker` struct:

```
Go   
type Worker struct {
    // ... existing fields ...
    taskOutputs      chan taskOutput
    retryChan        chan taskOutput // Add this
    // ... existing fields ...
}
```

### Step 2: Initialize it in `main`

Don't forget to initialize it where you create the worker:

```
Go   
worker := Worker{
    hydfsClient:   hydfsClient,
    done:          make(chan bool),
    tasks:         make(map[TaskID]*localTask),
    taskOutputs:   make(chan taskOutput, 100),
    retryChan:     make(chan taskOutput, 100), // Initialize with buffer
    // ...
}
```

### Step 3: Update the Consumer Loop (The `select`)

Modify the tuple sending goroutine (starting around line 144) to listen to **both** channels. This ensures that even if `taskOutputs` is full of new data, the consumer can still process retries.

```
Go   
// Inside the goroutine for sending out tuples
go func() {
    defer wg.Done()
    for {
        select {
        case <-ctx.Done():
            return
        case t := <-taskOutputs:
            // Process tuple t
        }
    }
}
```

```
// Case 1: Prioritize Retries (or treat them equally)
case out := <-worker.retryChan:
    worker.processTupleOutput(ctx, out) // Extract logic to helper function
// Case 2: Standard Output
case out := <-worker.taskOutputs:
    worker.processTupleOutput(ctx, out)
}
}
}()
```

#### Step 4: Update the Retry Logic

Change your error handling code to push to `retryChan` instead of `taskOutputs`.

Go



```
// Inside your processTupleOutput logic (or wherever the send logic is now):
if err != nil {
    _ = client.Conn.Close()
    worker.connectionsLock.Lock()
    delete(worker.connections, nextWorker)
    worker.connectionsLock.Unlock()

    // FIX: Send to retryChan instead of taskOutputs
    go func(t taskOutput) {
        time.Sleep(100 * time.Millisecond)
        select {
        case worker.retryChan <- t:
            // Successfully queued for retry
        case <-ctx.Done():
            // Worker is shutting down, stop retrying
        }
    }(out)
    return // or continue
}
```

#### Why this works

By using a separate channel, the **Task Producers** can block on `taskOutputs` (when it's full) without preventing the **Consumer** from reading from `retryChan`. The `select` statement guarantees that the consumer will continue to drain the retries, breaking the deadlock and allowing the system to recover.