Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Principios de Sistemas Operativos

MOTOR DE PROCESAMIENTO DISTRIBUIDO DESDE CERO

Ruta A: Mini-Spark (Batch DAG)

Profesor: Kenneth Obando Rodriguez

Estudiante: Kendall Josue Piedra Navarro

Carné: 2023164712

Fecha: 05 de Diciembre, 2025

Índice

1. Introducción

El presente documento detalla el diseño y la implementación de un sistema distribuido mínimo, siguiendo la **Ruta A: Batch DAG (mini-Spark)** del Proyecto 2 del curso Principios de Sistemas Operativos. El objetivo fundamental ha sido ejercitar conceptos clave como planificación, concurrencia (procesos/hilos), IPC y coordinación distribuida.

El sistema opera bajo una arquitectura **Maestro-Esclavo (Master/Coordinator)** y **Workers**, diseñada para procesar trabajos por lotes sobre archivos de datos, ejecutando un **Grafo Dirigido Acíclico (DAG)** de operaciones (Map, Filter, Reduce) sobre grandes conjuntos de datos.

1.1. Visión General del Proyecto

El **Motor de Procesamiento Distribuido Batch (Mini-Spark)** es un sistema multinodo que prioriza:

- **Escalabilidad horizontal:** Capacidad de añadir más Workers para aumentar la capacidad de procesamiento.
- **Eficiencia en la comunicación IPC:** Uso de HTTP/1.1 para todas las comunicaciones inter-proceso.
- **Tolerancia a fallos:** Replanificación automática de tareas mediante detección de fallos con Heartbeats.

1.2. Patrón Arquitectónico: Maestro-Esclavo

La arquitectura se fundamenta en el patrón **Maestro-Esclavo**, priorizando la escalabilidad horizontal y la tolerancia a fallos.

El sistema se compone de tres entidades principales:

- **Master/Coordinator:** Actúa como el **Coordinador** y el **Cerebro**. Su rol es gestionar el flujo de control, la planificación, el monitoreo del estado y la tolerancia a fallos. **No procesa datos** ni participa en la transferencia de datos masivos. Es responsable del registro de Workers, la recepción de Jobs y la planificación de tareas (Scheduling).
- **Workers:** Actúan como los **Ejecutores** y el **Músculo**. Su rol es ejecutar el procesamiento real de los datos (Map, Reduce) y gestionar su propio almacenamiento local y comunicación con otros Workers. Son nodos de ejecución que procesan las tareas de forma aislada, gestionan particiones de datos y reportan su estado. Son **homogéneos** en capacidades.
- **Cliente (CLI):** Interfaz para enviar Jobs y consultar su estado a través de la API.

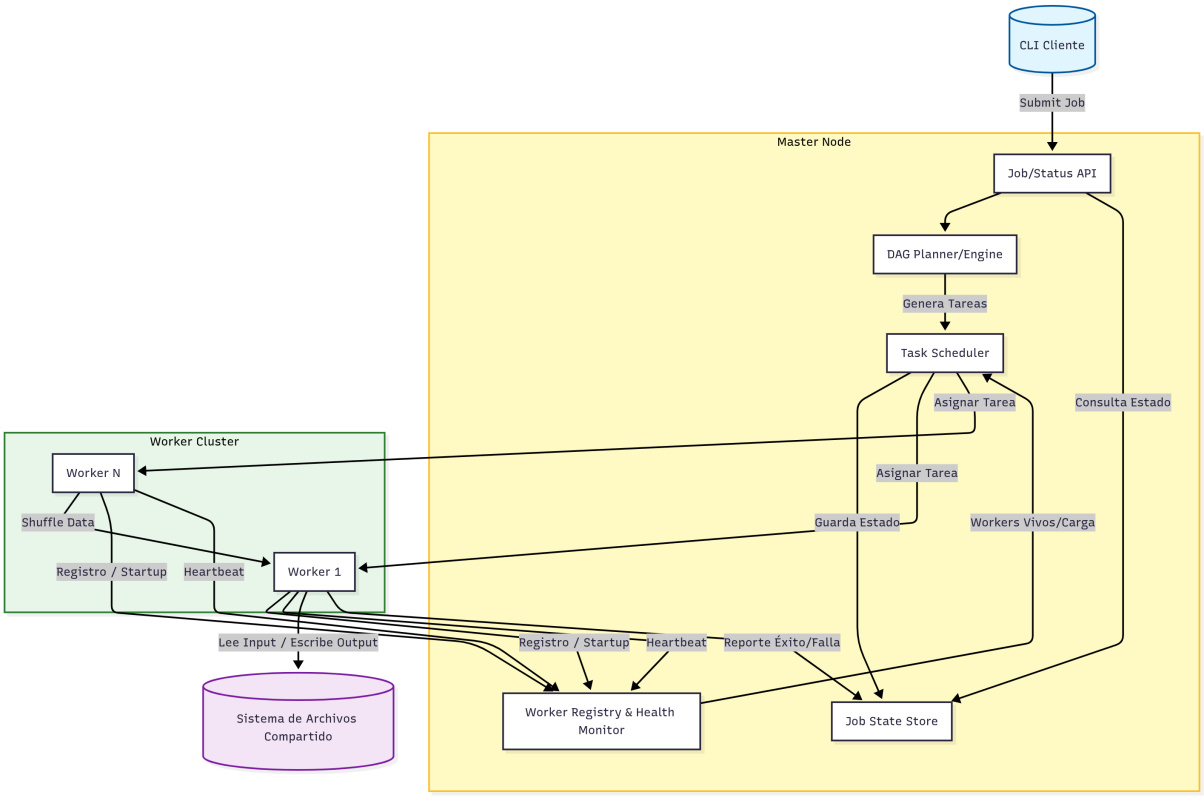

Figura 1: Diagrama General de Componentes y Flujo de Comunicación

2. Diseño Estático del Cluster

2.1. Estructura de Componentes

La arquitectura lógica se compone de un Master único y un clúster de N Workers replicables. La Figura 1 visualiza esta estructura.

2.1.1. Componentes Internos del Master

El Master se compone de módulos internos dedicados a la coordinación:

1. **API Server (HTTP):** Única interfaz para recibir peticiones del Cliente y reportes de los Workers.
2. **DAG Engine:** Analiza el `JobRequest`, realiza el **Ordenamiento Topológico (Topological Sort)** del DAG y descompone el Job en *Tasks* atómicas mediante el ordenamiento topológico.
3. **Worker Registry & Health Monitor:** Mantiene el estado en tiempo real de los Workers a través del protocolo Heartbeat, esencial para la detección de fallos.
4. **Task Scheduler:** Asigna las Tareas a los Workers disponibles (usando Round-Robin) y gestiona la cola de replanificación.
5. **Job State Store (SQLite):** Persiste los metadatos y el estado de los Jobs y Tareas para recuperación (implementado con persistencia mínima en memoria para esta versión).

2.1.2. Componentes Internos del Worker

Cada Worker es una instancia autónoma con las siguientes responsabilidades:

1. **Task Executor Pool (Goroutines):** Un pool de hilos ligeros (Goroutines) para ejecutar concurrentemente las operaciones asignadas. Las tareas asignadas se envían a este pool para ser ejecutadas concurrentemente utilizando canales como semáforos, asegurando que el número de tareas activas sea limitado.
2. **Data Server (Shuffle Service):** Un servidor HTTP interno para exponer los datos intermedios que este Worker ha procesado a otros Workers. Permite que otros Workers realicen peticiones GET y descarguen la porción de datos necesaria para su operación Reduce o Join.
3. **Memory Manager (ExecutionManager):** Gestiona la memoria local para el *caching* de datos intermedios y el mecanismo de **Spill to Disk** (Desbordamiento a Disco) cuando se alcanza un umbral de memoria.

3. Protocolo IPC, API y Health Check

La comunicación inter-nodo (IPC) se realiza exclusivamente mediante **HTTP/1.1** con serialización **JSON**, facilitando la depuración y el diseño de la API.

3.1. Flujo de Control de un Job

La siguiente secuencia detalla la interacción entre los componentes para una ejecución completa que incluye la etapa crítica de **Shuffle**:

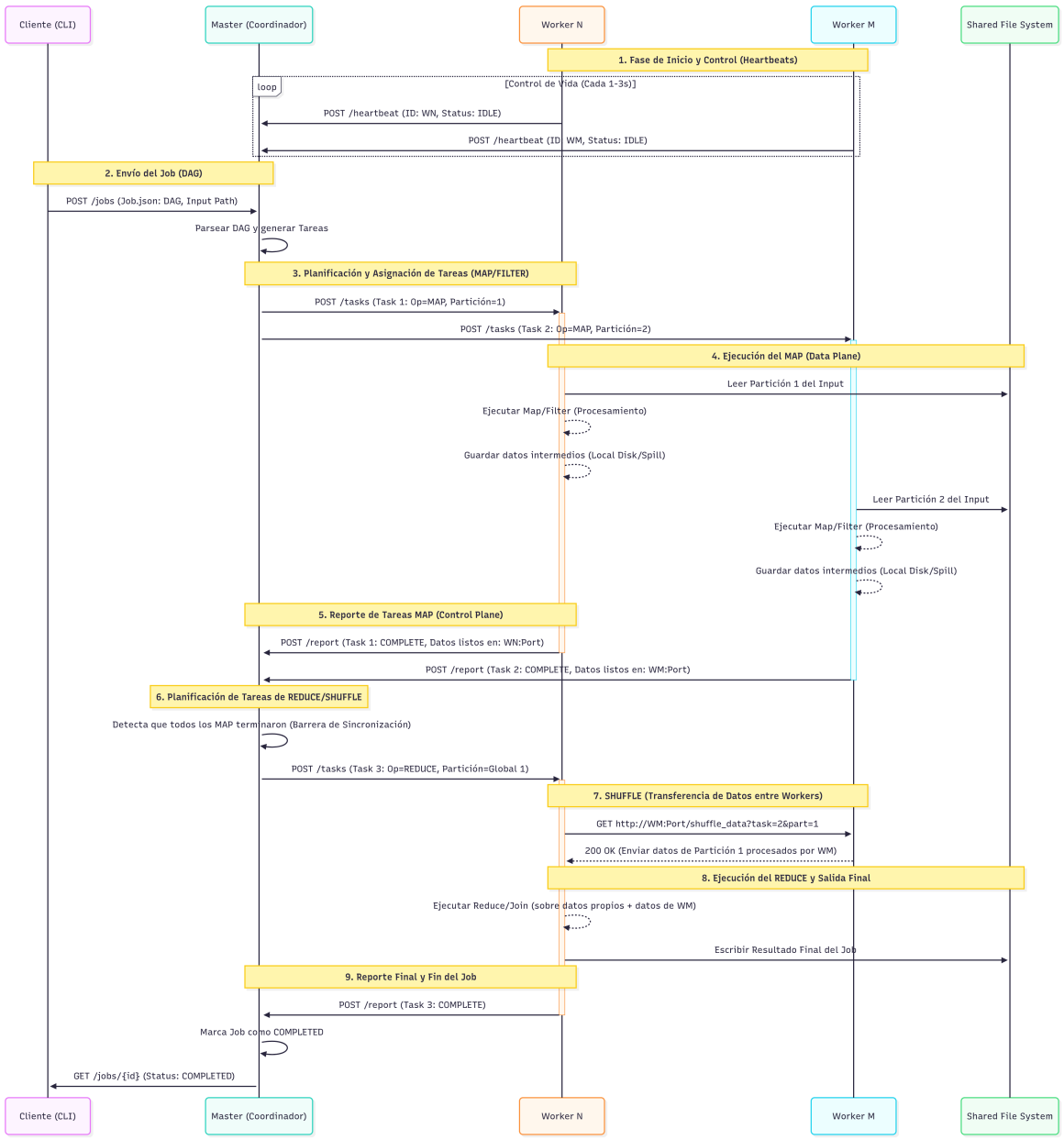

Figura 2: Diagrama de Secuencia y Flujo de Control

3.2. Especificación de APIs

3.2.1. Protocolo Cliente-Master

- **Envío de Job:** `POST /api/v1/jobs`
  - Cuerpo: Definición del Job en formato JSON
  - Respuesta: ID del Job asignado
- **Consulta de Estado:** `GET /api/v1/jobs/{id}`
  - Respuesta: Estado actual del Job (PENDING, RUNNING, COMPLETED, FAILED)

3.2.2. Transferencia de Datos P2P (Shuffle)

La comunicación de datos entre Workers utiliza la propia red de Worker a Worker (P2P). Cada Worker ejecuta un **Servidor HTTP** interno (`/data/shuffle`) para que otros Workers puedan realizar peticiones `GET` y descargar la porción de datos necesaria para su operación Reduce o Join.

Justificación: Si el Master intentara reenviar los datos, se convertiría en un cuello de botella masivo de ancho de banda. La comunicación P2P distribuye la carga de transferencia de datos a través de la red del clúster.

3.3. Protocolo Heartbeat (Registro de Workers)

El mecanismo de Heartbeat es la base de la tolerancia a fallos. Se utiliza un modelo de **concesión (Leasing)** donde la ausencia de la señal indica un fallo.

3.3.1. Especificación del Endpoint

Detalle	**Especificación**
Endpoint	`POST /heartbeat`
Frecuencia	Cada 3 ± 1 segundos (Worker $\rightarrow$ Master)
Timeout de Fallo	**10 segundos** (Master marca como DEAD)

Cuadro 1: Especificación del Protocolo Heartbeat

3.3.2. Modelo de Datos Heartbeat (Body JSON)

Este es el contrato API para el monitoreo:

```
{
  "worker_id": "worker-04",
  "address": "192.168.1.10:8081",
  "status": "IDLE",
  "active_tasks": 0,
  "mem_usage_mb": 512,
  "last_heartbeat": 1709214400
}
```

3.3.3. Lógica de Replanificación

1. **Detección de Fallos:** Si el Master no recibe un Heartbeat dentro del timeout de 10 segundos, el Worker Registry marca el `WorkerID` como `DEAD`.

2. **Activación de Rescate:** El Task Scheduler consulta el Job State Store y las tareas previamente asignadas al Worker caído son re-encoladas inmediatamente en la cola de `PENDING`.

3. **Recuperación:** Las tareas son reasignadas inmediatamente a Workers vivos (homogéneos) disponibles para continuar la ejecución del Job.

4. Decisiones de Sistemas Operativos (SO)

Esta sección justifica las elecciones de bajo nivel que optimizan el rendimiento, el uso de recursos y la escalabilidad del sistema.

4.1. Principio de Shared-Nothing

El diseño se adhiere al principio **Shared-Nothing**, donde los nodos operan de forma independiente y no comparten memoria o almacenamiento central de datos brutos.

- **Justificación:** Elimina el cuello de botella central en el Master. Garantiza que la latencia y el rendimiento no empeoren al añadir más Workers.
- **Implicación de SO:** El Master **no accede** al Sistema de Archivos de Input/Output. Solo los Workers tienen montado el volumen de datos. El Master solo maneja las **rutas lógicas** de los archivos.

4.2. Modelo de Concurrencia (Goroutines M:N)

El proyecto utiliza un modelo de concurrencia basado en el paradigma multi-proceso para los nodos y **multi-hilo ligero** (Goroutines) para la ejecución interna de tareas.

- **Master y Workers:** Se ejecutan como procesos independientes. El Master se encarga únicamente de orquestar Workers y asignarles tareas.
- **Modelo M:N:** El Worker utiliza el modelo de concurrencia nativo de Go, donde muchas **Goroutines** (M) son planificadas sobre un conjunto limitado de **hilos del SO (OS Threads)** (N). En lugar de mapear cada tarea a un hilo del SO (modelo 1:1), Go utiliza este modelo M:N más eficiente.
- **Justificación de SO:** Las Goroutines son mucho más ligeras y rápidas en la creación y el cambio de contexto (*context switching*) que los hilos tradicionales. Esto permite al Task Executor Pool manejar miles de tareas concurrentes con un *overhead* de memoria y CPU mínimo, maximizando la utilización de los núcleos del procesador.
- **Worker Pool (ExecutionManager):** Cada Worker mantiene un **Pool de Hilos** (Goroutines). Las tareas asignadas se envían a este pool para ser ejecutadas concurrentemente utilizando canales como semáforos, asegurando que el número de tareas activas sea limitado.
- **Control Loop:** La lógica del Scheduler en el Master opera en una goroutine separada (`ControlLoop`) que se ejecuta periódicamente (`time.Ticker`) para evitar **bloquear el hilo principal** de la API, manteniendo al Master dedicado a la coordinación.

5. Planificación y Gestión de Recursos

5.1. Planificación de Tareas (Scheduler)

El Planificador gestiona el avance del DAG (Grafo Dirigido Acíclico) y la asignación de tareas.

- **Avance del DAG:** El Scheduler utiliza un enfoque basado en etapas. Una etapa (`stage`) solo avanza a la siguiente (e.g., Map → Reduce) cuando **todas** sus tareas han reportado éxito.
- **Política de Asignación:** Se utiliza una política de **Round-Robin** simple para distribuir la carga entre Workers. Se incluye una consideración básica de **carga** al ignorar Workers que estén `DOWN` o reporten saturación de tareas activas.

5.2. Gestión de Memoria y Almacenamiento (Spill-to-Disk)

La gestión de memoria implementa un mecanismo de **backpressure** a nivel de SO, vital para la etapa de Shuffle y para cumplir con el requisito de manejo de memoria.

- **Cache en Memoria:** El `MemoryAggregator` del Worker almacena pares clave-valor (K/V) recibidos del Shuffle en un `map` de Go para la agregación.
- **Mecanismo de Spill-to-Disk:** El Memory Manager monitorea la memoria consumida por los resultados intermedios de las operaciones Map. Si el tamaño total del `map` excede un umbral configurable (e.g., 50 MB o `MAX_MEM`), el Worker detiene temporalmente la ejecución y el `MemoryAggregator` serializa su contenido a archivos temporales (`/tmp/spill_...jsonl`), vaciando la memoria.
- **Decisión de SO:** Esto evita que el proceso Worker se quede sin memoria y colapse, actuando como una válvula de seguridad del sistema operativo.
- **Persistencia de Resultados:** Los resultados finales del Job se escriben en una ruta de almacenamiento local persistente (ej: `./data/outputs/`) definida por el Master al crear la tarea, cumpliendo con el requisito de almacenamiento de salida.

6. Tolerancia a Fallos y Resiliencia

El sistema implementa mecanismos mínimos de tolerancia a fallos para sobrevivir a la pérdida de un nodo de ejecución.

- **Detección de Fallos (Heartbeats):** El Master utiliza el mecanismo de **Heartbeats**. Si un Worker no reporta su estado dentro de un `WorkerTimeoutSeconds` de **10 segundos**, el Master Registry lo marca como `DOWN`.
- **Replanificación (Reintento de Tarea):** Las tareas que estaban asignadas a un Worker marcado como `DOWN` son **re-encoladas** inmediatamente en la cola de tareas pendientes para ser reasignadas al próximo Worker vivo disponible.
- **Límite de Reintentos:** Cada tarea tiene un contador de reintentos (`RetryCount`). Si una tarea falla (por error de código o fallo del Worker) y su contador supera el `MaxTaskRetries` (configurado en 3), el Job es marcado como `FAILED`.
- **Idempotencia Básica:** Se implementa una idempotencia básica para evitar duplicar resultados en reejecuciones, gestionando el ID del intento de tarea.

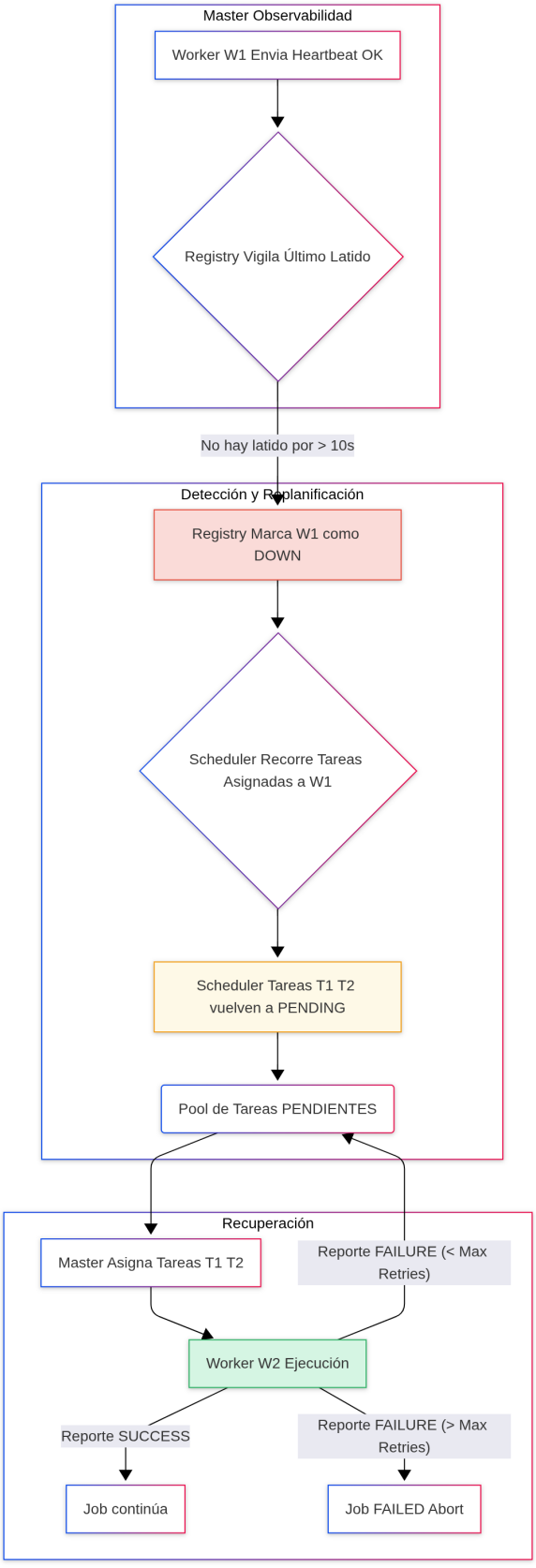

Figura 3: Diagrama del Flujo de Tolerancia a Fallos y Replanificación

7. Métricas y Observabilidad

Se implementó un esquema de **Observabilidad** basado en logs estructurados y métricas reportadas para facilitar el monitoreo y diagnóstico del sistema.

- **Métricas por Nodo (Worker):** Cada Heartbeat incluye métricas básicas como:
  - Uso de CPU y memoria
  - Número de tareas activas
  - Latencia promedio de procesamiento
- **Métricas por Job:** Se registra:
  - Tiempo total de ejecución
  - Etapas completadas
  - Número de fallos y reintentos
- **Logging:** Uso de la librería estándar `log` de Go con prefijos para identificar el componente (Master, Worker, Scheduler), facilitando el diagnóstico y la depuración del sistema distribuido.

8. Conclusiones

El sistema Mini-Spark implementado demuestra los principios fundamentales de un motor de procesamiento distribuido:

1. La arquitectura Maestro-Esclavo proporciona una clara separación de responsabilidades entre coordinación (Master) y ejecución (Workers).
2. El uso de Goroutines y el modelo M:N permite una concurrencia eficiente con bajo overhead.
3. El principio Shared-Nothing elimina cuellos de botella centralizados y permite escalabilidad horizontal.
4. Los mecanismos de tolerancia a fallos mediante Heartbeats y replanificación automática proporcionan resiliencia ante fallos de nodos.
5. La gestión de memoria con Spill-to-Disk previene fallos por agotamiento de recursos.

Este diseño proporciona una base sólida para comprender los conceptos de sistemas distribuidos aplicados en motores de procesamiento de datos modernos como Apache Spark.