# Mini-Spark: Arquitectura del Sistema

## 1. Visión General

Mini-Spark es un motor de procesamiento distribuido por lotes (*Batch Processing*) diseñado bajo la arquitectura **Master-Worker**. El sistema permite la ejecución de grafos acíclicos dirigidos (DAGs) de operaciones de transformación de datos (Map, Reduce, Join, etc.) sobre un clúster de nodos coordinados.

El objetivo principal es demostrar conceptos de sistemas operativos como concurrencia, comunicación entre procesos (IPC), gestión de memoria y sistemas de archivos distribuidos.

## 2. Diseño de Arquitectura

El sistema consta de tres componentes principales desacoplados que se comunican a través de HTTP/JSON.

### 2.1 Componentes

1. **Master (Coordinador):**
   - Es el cerebro del sistema.
   - Mantiene el estado global del clúster (Workers activos) y de los trabajos (Jobs).
   - Contiene el **Planificador (Scheduler)** que decide qué tarea ejecutar y dónde.
   - Expone una API REST para el Cliente y puntos de control para los Workers.
2. **Worker (Trabajador):**
   - Ejecuta las tareas asignadas (cómputo pesado).
   - Gestiona la I/O (Lectura/Escritura) de archivos.
   - Reporta su salud mediante *heartbeats* periódicos.
   - Ejecuta operadores definidos por el usuario (UDFs).
3. **Cliente (CLI):**
   - Interfaz de línea de comandos para enviar trabajos y consultar estado.
   - Valida la estructura del DAG antes de enviarlo.

### 2.2 Modelo de Datos (Shared Filesystem)

Dado que es un sistema distribuido, asumimos un sistema de archivos compartido (simulado mediante volúmenes Docker montados en /tmp/mini-spark). Esto permite que el Worker A escriba un resultado intermedio que luego el Worker B puede leer.

## 3. Modelo de Procesos e Hilos

El sistema utiliza el modelo de concurrencia nativo de Go (Goroutines) para manejar la ejecución paralela y asíncrona.

### 3.1 Master

- **Hilo Principal:** Inicia el servidor HTTP.
- **Goroutines HTTP:** Una por cada petición entrante (API non-blocking).
- **Scheduler Loop (Goroutine):** Bucle infinito que monitorea la cola de tareas pendientes y busca workers libres.
- **HealthCheck Loop (Goroutine):** Verifica cada 5 segundos si los workers han enviado latidos recientes.

### 3.2 Worker

- **Hilo Principal:** Servidor HTTP para recibir comandos.
- **Heartbeat Loop (Goroutine):** Envía un POST al Master cada 3 segundos con métricas.
- **Task Executor (Goroutines):** Cada tarea recibida se lanza en una goroutine independiente, permitiendo paralelismo dentro del mismo nodo (aunque limitado por el scheduler para no saturar).

## 4. Protocolos de Comunicación

Toda la comunicación es **HTTP/1.1** transportando cargas útiles en **JSON**.

### 4.1 API Pública (Cliente -> Master)

| Método | Endpoint | Descripción |
|---|---|---|
| POST | /api/v1/jobs | Recibe un JSON con la definición del DAG. |
| GET | /api/v1/jobs/{id} | Devuelve estado, progreso (%) y métricas. |
| GET | /api/v1/jobs/{id}/results | Devuelve las rutas de los archivos finales. |

### 4.2 Protocolo Interno (Master <-> Worker)

| Dirección | Endpoint | Payload | Propósito |
|---|---|---|---|
| **W -> M** | /register | {port, id} | Worker se une al clúster. |
| **W -> M** | /heartbeat | {metrics: {cpu, ram}} | Señal de vida y carga. |

| | | | |
|---|---|---|---|
| **M -> W** | /task | {op, input, fn} | Asignación de tarea. |
| **W -> M** | /task/complete | {status, result_path} | Reporte de finalización. |

# 5. Planificación y Ejecución (Scheduler)

El planificador implementa una lógica basada en dependencias de grafos.

1. **Recepción:** Al recibir un Job, se calcula el *in-degree* (número de padres) de cada nodo.
2. **Cola Inicial:** Los nodos con in-degree == 0 (generalmente Read) se colocan en la cola TaskQueue.
3. **Asignación:** El bucle del planificador extrae tareas y las asigna a los Workers usando una política **Round-Robin** filtrando solo nodos con estado UP.
4. **Transición:**
    - Cuando una tarea termina exitosamente, el Master actualiza el grafo.
    - Verifica los hijos de esa tarea. Si un hijo tiene **todas** sus dependencias satisfechas, se mueve a la TaskQueue.

# 6. Gestión de Memoria y Almacenamiento

## 6.1 Spill to Disk (Volcado a Disco)

Para operaciones de agregación (ReduceByKey) que requieren mantener estado en memoria (Hash Maps), implementamos un mecanismo de protección:

- **Umbral Configurable:** SpillThreshold (por defecto 1000 claves).
- **Comportamiento:** Si el mapa en memoria supera el umbral, el Worker:
    1. Serializa el mapa actual a un archivo temporal (spill_X.tmp).
    2. Limpia la memoria RAM.
    3. Continúa procesando.
- **Merge Final:** Al terminar la lectura, el Worker fusiona los archivos temporales con lo que quede en memoria para generar el resultado final.

## 6.2 Persistencia del Master

El Master guarda su estado crítico (Jobs, Progress, Outputs) en un archivo local master_state.json cada vez que ocurre un evento importante (Job recibido, Tarea completada). Esto permite recuperar el clúster tras un reinicio inesperado.

# 7. Tolerancia a Fallos

El sistema implementa **Alta Disponibilidad (HA)** básica mediante detección y recuperación

activa.

### 7.1 Detección de Fallos

- El Master mantiene una tabla de tiempos de LastHeartbeat.
- Si Time.Now() - LastHeartbeat > 10s, el worker se marca como DOWN.

### 7.2 Recuperación

1. El Master identifica todas las tareas que estaban asignadas al worker muerto (RunningTasks).
2. Esas tareas se marcan como fallidas y se re-encolan (TaskQueue).
3. En el siguiente ciclo, el Scheduler asignará esas tareas a un worker vivo diferente.

### 7.3 Reintentos

Si una tarea falla por error lógico (código de retorno del worker) y no por muerte del nodo, el Master la reintenta hasta **3 veces** (MaxRetries) antes de marcar el Job completo como fallido.

## 8. Conclusiones

La arquitectura implementada satisface los requisitos de un sistema distribuido robusto:

- **Escalabilidad:** Se pueden agregar N workers dinámicamente.
- **Resiliencia:** Soporta caídas de nodos sin detener el trabajo.
- **Observabilidad:** Provee métricas claras al usuario.
- **Eficiencia:** Gestiona recursos limitados mediante *Spill to Disk*.