# Mini-Spark: Arquitectura, Diseño e Implementación

Sistema distribuido tipo MapReduce/cluster de procesamiento en Rust

Autor: *(Nombre del estudiante)*

Curso: Principios de Sistemas Operativos

Proyecto 2

4 de diciembre de 2025

## Índice

# 1. Introducción

Mini-Spark es un sistema distribuido de procesamiento batch inspirado en Apache Spark pero diseñado para ser lo bastante compacto como para caber en un proyecto de curso, y lo bastante completo como para cubrir conceptos reales de sistemas operativos y sistemas distribuidos: procesos, hilos, planificación, memoria, fallos, persistencia y observabilidad.

El proyecto permite ejecutar trabajos (jobs) de procesamiento sobre archivos de texto, aplicando una cadena de operadores (`map`, `filter`, `flat_map`, `reduce_by_key`, `join`, `shuffle_write`) definidos en un DAG (grafo acíclico dirigido). Ese DAG se rompe en tareas físicas (*tasks*) que se distribuyen entre múltiples procesos worker. El master coordina el sistema, el client expone una CLI amigable, y la infraestructura de scripts `.sh` automatiza pruebas E2E, tolerancia a fallos, métricas, balanceo de carga y benchmarks.

Esta documentación describe la arquitectura completa del sistema, haciendo énfasis en:

- Modelo de procesos e hilos.
- API pública (REST + CLI).
- Protocolos internos (registro, heartbeat, asignación de tareas, completado).
- Planificación y balanceo de carga.
- Gestión de memoria y cache con spill a disco.
- Manejo de fallos y reintentos.
- Persistencia de estado de jobs.
- Métricas, observabilidad y scripts de prueba.

No se usan estos puntos como listas aisladas; cada apartado se desarrolla en detalle, mezclando explicación narrativa con algunos listados sólo cuando ayudan a clarificar.

# 2. Arquitectura general

## 2.1. Componentes

La arquitectura lógica se basa en tres procesos principales:

**Master** Servicio central que:

- Mantiene el estado de jobs, tareas y workers.
- Expone la API HTTP para workers y cliente.
- Planifica tareas usando un scheduler round-robin con *load awareness*.
- Persiste el estado de jobs a disco.
- Expone métricas del sistema, jobs y fallos.

**Workers** Cada worker es un proceso que:

- Se registra en el master y mantiene heartbeats periódicos.
- Ejecuta en paralelo tareas sobre particiones.
- Mantiene una cache de particiones con límite de memoria y spill a disco.
- Expone un endpoint `/metrics` con sus métricas locales.

**Client** Proceso efímero que se ejecuta por comando para:

- Enviar jobs (wordcount, join, etc.).
- Consultar estado y resultados.
- Consultar métricas del master.

### 2.2. Topología y despliegue

La topología típica del clúster es:

- Un `master` escuchando en `0.0.0.0:8080`.
- Varios `workers` en puertos como `9000`, `9010`, `9020`, configurados vía variables de entorno (`WORKER_PORT`).
- Cada worker expone métricas en `WORKER_PORT + 1000` (p. ej. `10000`).
- El cliente apunta al master usando `MASTER_URL`.

El sistema se puede desplegar de dos formas:

**Modo nativo** Ejecutando directamente los binarios `./target/release/master`, `worker` y `client`, apoyados en scripts como `test_single_node.sh`, `test_fault_tolerance.sh`, etc.

**Modo Docker** Usando `docker-compose`. Scripts como `docker-demo.sh`, `docker-fault-test.sh` y `benchmark_1m.sh` levantan el clúster completo y ejecutan demostraciones o benchmarks.

En ambos modos, el directorio `/tmp/minispark` (o `C:/tmp/minispark` en Windows) se utiliza para almacenar estado, particiones y archivos de spill.

## 3. Modelo de procesos, hilos y concurrencia

### 3.1. Procesos y responsabilidad

Se siguen las ideas clásicas de sistemas operativos:

- **Un proceso master** por clúster: responsable de coordinación global.
- **Varios procesos worker**: uno por instancia de procesamiento.
- **Procesos cliente**: se crean bajo demanda para las operaciones de CLI.

Cada script de prueba coordina varios procesos simultáneos. Por ejemplo, `test_metrics.sh` arranca un master, un worker y luego un cliente que envía un job mientras se consultan métricas por HTTP.

### 3.2. Master: tareas asíncronas y acceso concurrente

El master está construido con `tokio` y `axum`, por lo que internamente utiliza un pool de hilos de *runtime* asíncrono.

Sobre ese runtime se ejecutan varias tareas lógicas:

1. El servidor HTTP que atiende:
   - `/api/v1/jobs`, `/api/v1/workers/*`, `/api/v1/metrics/*`, etc.

2. Una tarea de **monitorización de workers**, que revisa periódicamente los heartbeats y marca workers como `Down` si no responden.

3. Una tarea de **reintentos**, que vacía la cola de tareas fallidas y crea nuevos intentos hasta un máximo configurado.

4. Una tarea de **persistencia periódica**, que llama a `PersistenceManager::flush()` para escribir el estado a disco con cierta frecuencia.

El estado compartido (jobs, tareas, workers, métricas) está encapsulado en una estructura `AppState` protegida con `Arc<Mutex<...>>`. Esto garantiza exclusión mutua a pesar de la concurrencia entre peticiones HTTP y tareas de fondo.

### 3.3. Workers: pool lógico de ejecutores

Cada worker se ejecuta como un proceso independiente pero internamente lanza varias tareas de `tokio`:

- Una tarea de **heartbeat**, que cada `HEARTBEAT_INTERVAL_SECS` segundos envía al master un objeto `Heartbeat{worker_id, active_tasks}`.
- Una tarea para el **servidor de métricas**, que expone el endpoint `/metrics` en `WORKER_PORT + 1000`.
- Un conjunto de tareas ejecutoras, tantas como indique `WORKER_THREADS`. Cada ejecutor:
    1. Llama a `/api/v1/workers/task` para pedir trabajo.
    2. Ejecuta el operador sobre la partición asociada.
    3. Envía el resultado al master via `/api/v1/workers/task/complete`.

Aunque se llaman "threads" en la configuración, realmente son *tasks* asíncronas dentro del runtime `tokio`. Sin embargo, desde el punto de vista de diseño de sistemas operativos, se comportan como un pool de hilos cooperativos que comparten el proceso worker.

Un `AtomicU32` mantiene el número actual de tareas activas. Este contador se incrementa cuando un ejecutor recibe una tarea y se decrementa al completarla. Esa información se usa tanto en heartbeats como en métricas de carga por worker.

### 3.4. Límite de tiempo por tarea

Para evitar que una tarea colgada bloquee recursos indefinidamente, el worker establece `MAX_TASK_TIME_SECS`. Tras ejecutar el operador, se mide la duración con `Instant::now()`:

- Si el tiempo excede el límite, el worker marca la tarea como fallida aunque no haya ocurrido un error lógico.
- Se envía al master un `TaskResult` con `success = false` y un mensaje de error indicando que se superó el límite.

Esto conecta directamente con los conceptos de watchdogs y timeouts vistos en sistemas operativos.

# 4. Modelo de datos y DAG de ejecución

## 4.1. Representación de jobs y tareas

En el módulo `common` se define la estructura central `JobInfo`. De forma simplificada, un job contiene:

- Un identificador `id: Uuid`.
- El `JobRequest` original, con:
  - Nombre del job.
  - DAG lógico de operadores.
  - Paralelismo (número de particiones).
- Estado del job (`JobStatus`: Accepted, Running, Succeeded, Failed).
- Progreso en % y timestamps de creación/actualización.
- Conjuntos de tareas pendients, corriendo y completadas.

Cada `Task` representa la ejecución de un nodo del DAG sobre una partición específica. Incluye:

- `id`, `job_id`, `node_id`.
- `partition_id` y `num_partitions`.
- Operación (`op`) y parámetros específicos (p. ej. claves para join).
- Rutas de entrada (`input_paths`) o salidas de shuffles previos.
- Campo `attempt` para los reintentos.

## 4.2. DAG lógico a plan físico

El usuario no define tareas, sino un DAG lógico de operadores. El master se encarga de traducir ese DAG en un conjunto de tareas físicas mediante una función de planificación de etapas que:

1. Analiza las dependencias entre nodos del DAG.
2. Inserta nodos `shuffle_write` cuando una operación requiere barajar datos por clave (`reduce_by_key`, `join`).
3. Para cada nodo y cada partición, genera una `Task` con:
   - Identificador único.
   - Referencias al job y al nodo.
   - Rutas de archivos intermedios bajo `/tmp/minispark`.

Desde el punto de vista del master, el DAG se convierte en un pipeline de etapas (stages). Cada stage agrupa las tareas que pueden ejecutarse en paralelo una vez que sus dependencias han finalizado.

### 4.3. Particiones y formato de datos

Los datos se manejan como **particiones** definidas por la estructura `Partition`, que contiene un vector de `Record`. Cada `Record` almacena:

- `key: Option<String>` (para operaciones por clave).
- `value: String` (la línea de texto o valor transformado).

Las particiones se serializan a JSON en archivos de la forma:

`/tmp/minispark/{job_id}_{node_id}_p{partition_id}.json`

y se reutilizan entre tareas como entradas de operadores posteriores, o se cargan en cache en los workers para acelerar lecturas repetidas.

## 5. API pública del sistema

### 5.1. API HTTP del master

El master implementa un servicio REST con endpoints que se agrupan en tres categorías: *control de workers*, *gestion de jobs* y *métricas/estado*.

#### Control de workers

- `POST /api/v1/workers/register`
  El worker se registra enviando un objeto `WorkerRegistration`. El master asigna un UUID si el `id` es nulo y devuelve el registro actualizado.
- `POST /api/v1/workers/heartbeat`
  Recibe un `Heartbeat{worker_id, active_tasks}`. Actualiza el estado del worker, su contador de tareas activas y la marca temporal de último heartbeat.
- `GET /api/v1/workers/task?worker_id=...`
  El worker pide una tarea. El master selecciona una tarea ready y apropiada según el scheduler y la devuelve en JSON. Si no hay trabajo, devuelve `204 No Content`.
- `POST /api/v1/workers/task/complete`
  Recibe un `TaskResult` con éxito o fallo. El master actualiza el estado de la tarea, ajusta el progreso del job, procesa outputs de shuffle y dispara reintentos o fallos de job si corresponde.

#### Gestión de jobs

- `POST /api/v1/jobs`
  Punto de entrada para nuevos trabajos. Recibe un `JobRequest` con nombre, DAG y paralelismo. Devuelve un `JobInfo` simplificado con el `job_id` asignado.
- `GET /api/v1/jobs/{id}`
  Devuelve estado actual del job (estado, progreso, timestamps, etc.).
- `GET /api/v1/jobs/{id}/results`
  Lista rutas o contenidos de resultados finales para el job (por ejemplo, salida del wordcount).
- `GET /api/v1/jobs/{id}/metrics`
  Devuelve un resumen de métricas de ese job: número de tareas, tareas exitosas, fallidas, tiempos, etc.

- `GET /api/v1/jobs/{id}/stages`
  Devuelve estadísticas por etapa (stage) del DAG: tiempos de inicio/fin, número de tareas por stage, etc.

**Métricas y estado global**

- `GET /health`
  Endpoint de salud simple.
- `GET /api/v1/metrics/system`
  Métricas globales del sistema: número de workers (Up/Down), jobs en diferentes estados, tareas en cola/corriendo, cargas por worker, etc.
- `GET /api/v1/metrics/jobs`
  Lista de métricas de todos los jobs.
- `GET /api/v1/metrics/failures`
  Métricas de fallos por job, workers caídos, reintentos, etc.
- `GET /api/v1/metrics/scheduler`
  Información del scheduler: total de workers, activos, índice de round-robin, distribución de carga.
- `GET /api/v1/state`
  Devuelve el estado persistido de jobs (`PersistedState`) tal como está almacenado en `jobs.json`.

## 5.2. CLI del cliente

El cliente proporciona una interfaz de alto nivel que abstrae la API REST. Algunos comandos clave:

`submit` Envia un job de WordCount:

```
client submit --input data/input.csv --parallelism 4 --name wc-example
```

Imprime por consola el `JOB_ID` asignado.

`submit-join` Envia un job de join:

```
client submit-join --name join-multi \
  --parallelism 4 \
  --sales data/sales.csv \
  --products data/products.csv
```

`status JOB_ID` Consulta el estado de un job; muestra estado, progreso y algunos datos resumidos.

`results JOB_ID` Recupera y muestra resultados finales del job.

`metrics system` Golpea `/api/v1/metrics/system` y lo imprime formateado.

`metrics jobs` Muestra métricas de todos los jobs.

`metrics failures` Muestra contador de fallos del sistema.

`metrics job JOB_ID` Muestra métricas detalladas de un job concreto.

`metrics stages JOB_ID` Lista estadísticas de etapas para ese job.

Esta CLI se utiliza intensivamente en los scripts de prueba, que actúan como clientes automatizados para ejercitar la API de distintas formas.

### 5.3. Endpoint de métricas del worker

Cada worker expone un pequeño servidor HTTP que escucha en `WORKER_PORT + 1000` y soporta:

```
GET /metrics
```

La respuesta contiene un objeto `WorkerMetrics` que incluye:

- Identificador del worker y tiempo de actividad.
- Métricas de sistema (CPU, memoria usada, memoria total, porcentaje).
- Métricas de tareas (total ejecutadas, éxitos, fallos, tareas activas, latencias promedio/p50/p99).
- Métricas de cache (hits, misses, memoria de cache ocupada).

Esto permite inspeccionar la salud y carga de cada worker de forma independiente al master.

## 6. Protocolos internos

### 6.1. Registro de workers

Al iniciar, el worker construye un `WorkerRegistration` con:

- `id = Uuid::nil()` (el master asigna uno real).
- `host = "127.0.0.1"` u otra IP.
- `port = WORKER_PORT`.

Ese objeto se envía via `POST /api/v1/workers/register`. El master:

1. Asigna un UUID si el id es nulo.
2. Crea un `WorkerInfo` con ese id, estado `Up`, contador de tareas activas en 0 y `last_heartbeat = now()`.
3. Devuelve al worker un `WorkerRegistration` actualizado, que el worker guarda como su identida real.

### 6.2. Heartbeat y detección de workers caídos

El protocolo de heartbeat sigue una lógica simple pero eficaz:

- Cada `HEARTBEAT_INTERVAL_SECS` segundos, el worker envía:

  ```
  Heartbeat { worker_id, active_tasks }
  ```

- El master actualiza el `last_heartbeat` del worker, así como su campo `active_tasks`.
- Una tarea de fondo `monitor_workers` compara periódicamente el tiempo actual con `last_heartbeat`. Si se supera un umbral, se marca el worker como `Down`.

Cuando un worker pasa a `Down`, el master identifica las tareas que estaban asignadas a ese worker y aún no habían reportado resultado, las marca para replanificación y ajusta métricas de fallos.

### 6.3. Asignación de tareas a workers

El protocolo maestro–worker en torno a las tareas es:

1. El worker llama a:

   `GET /api/v1/workers/task?worker_id=...`

2. El master verifica si el worker existe y está `Up`. Si no, puede devolver `404` o `204`.
3. El master consulta el scheduler (ver Sección 7) para saber si el worker puede recibir carga adicional según su número de tareas activas.
4. Si puede, se selecciona una tarea `Pending` cuyo DAG ya esté listo (todas sus dependencias completadas). Se marca la tarea como `Running` y se asocia al worker.
5. Se devuelve la tarea en el cuerpo de la respuesta.
6. Si no hay tareas ready para ese worker, se responde `204 No Content`.

### 6.4. Reporte de completado de tareas

Cuando el worker termina una tarea, construye un `TaskResult` con campos como:

- `task_id`, `job_id`, `attempt`.
- `success: bool`.
- `error: Option<String>` (si falla).
- `output_path: Option<String>` (archivo final o intermedio).
- `records_processed: u64`.
- `shuffle_outputs: Vec<...>` (para `shuffle_write`).

Ese resultado se envía a:

`POST /api/v1/workers/task/complete`

El master valida que sea el intento más reciente para esa tarea (para evitar *duplicate completion*), actualiza el estado de la tarea y del job, y en caso de fallo decide si programar un reintento o marcar el job como fallido.

## 7. Planificación, scheduler y balanceo de carga

### 7.1. Política round-robin con conciencia de carga

El scheduler se basa en dos ideas combinadas:

1. **Round-robin**: los workers se recorren en un orden circular para repartir trabajo.
2. **Load awareness**: se evita asignar tareas a un worker cuya carga (tareas activas) sea muy superior a la del resto.

Para cada petición de tarea, el master consulta una estructura que mantiene:

- La lista de workers activos.

- El índice `next_index` para el siguiente worker en el orden lógico.
- Un máximo permitido de diferencia de carga entre workers (`max_load_diff`).

Si un worker tiene demasiadas tareas activas comparado con el menos cargado, la función `can_worker_receive_task` puede negar nuevas asignaciones, forzando a que el trabajo vaya a otros workers.

Esto se refleja en el endpoint `/api/v1/metrics/scheduler`, que muestra distribución de carga (`worker_loads`) y el `next_index` actual, permitiendo inspeccionar visualmente la equidad del reparto. El script `test_round_robin.sh` está precisamente diseñado para demostrar que la distribución es razonablemente equitativa.

### 7.2. Política de reintentos de tareas

Cada tarea tiene una cantidad máxima de reintentos (`MAX_TASK_ATTEMPTS`). El master distingue dos fuentes de fallo:

**Fallos lógicos o simulados** El worker ejecuta la tarea y devuelve `success = false`, bien sea por un error en el operador, una simulación mediante `FAIL_PROBABILITY`, o un timeout.

**Fallos por worker caído** El worker desaparece mientras tiene tareas asignadas; el master detecta este evento por falta de heartbeat, y reinyecta esas tareas.

En ambos casos, si el número de intentos no ha alcanzado el máximo, el master crea una nueva tarea idéntica pero con `attempt + 1` y la vuelve a insertar en la cola de pendientes. Si se agotan los reintentos, el job completo se marca como `Failed`, ya que no se garantiza que los datos intermedios estén en un estado coherente para seguir.

## 8. Gestión de memoria y cache en los workers

### 8.1. Diseño de la cache de particiones

Cada worker mantiene una estructura `PartitionCache` con un objetivo claro: almacenar particiones recientes en memoria para acelerar lecturas, pero respetando un límite de memoria configurable.

Internamente se gestiona:

- Un mapa `entries: HashMap<String, CacheEntry>` donde la clave representa `job_id:node_id:partitio`
- Un contador de memoria actual en bytes (`current_memory_bytes`).
- Un límite total en bytes (`max_memory_bytes`), derivado de `CACHE_MAX_MB`.

Cada `CacheEntry` incluye:

- La partición (`Partition`) o un valor vacío si fue *spilled*.
- Tamaño estimado en bytes.
- `last_access: Instant` para política LRU.
- Flags `spilled` y `spill_path`.

### 8.2. Spill a disco y política LRU

Cuando se inserta una partición vía `put` y la memoria actual supera el límite, la función `maybe_spill` entra en acción:

1. Se seleccionan entradas que no estén ya en disco.
2. Se ordenan implícitamente por `last_access` y se escoge la menos recientemente usada.
3. `spill_to_disk` serializa la partición a JSON en `/tmp/minispark/spill`, restando su tamaño al contador de memoria y dejando un `Partition::default()` en memoria.

La recuperación es transparente para el resto del código: si al hacer `get(key)` se encuentra que `spilled = true`, se lee el archivo de spill y se devuelve una partición reconstruida. Si hay algún error en la lectura, se registra y se considera un `miss`.

El script `test_cache.sh` está específicamente diseñado para:

- Forzar spills usando `CACHE_MAX_MB = 1`.
- Verificar existencia de archivos en `/tmp/minispark/spill`.
- Confirmar que la persistencia de estado de jobs convive correctamente con la lógica de cache.

### 8.3. Métricas de memoria y CPU del sistema

El `MetricsCollector` del worker incluye funciones para leer:

- `/proc/meminfo`: calcula memoria total (`MemTotal`) y disponible (`MemAvailable`), derivando memoria usada y porcentaje.
- `/proc/stat`: obtiene estadísticas de CPU y calcula una estimación del porcentaje de uso (user + nice + system frente a total).

Aunque es una aproximación (no se toman dos muestras separadas en el tiempo), es suficiente para fines de monitorización en este proyecto.

## 9. Manejo de fallos y tolerancia

### 9.1. Tipos de fallos considerados

Mini-Spark contempla varios tipos de fallos dentro del alcance del proyecto:

**Fallos de tarea** Errores al ejecutar un operador (p. ej. error de IO al leer una partición) o fallos simulados mediante la variable `FAIL_PROBABILITY`, que provoca devoluciones de `TaskResult.success = false`.

**Timeout de tarea** Una tarea puede ejecutarse correctamente pero superar el límite `MAX_TASK_TIME_SECS`. En ese caso el worker también la marca como fallida.

**Fallos de worker** Un worker puede "morir" (el proceso se mata o se cuelga). El master detecta este escenario cuando deja de recibir heartbeats dentro del intervalo esperado.

## 9.2. Reintentos por fallo de tarea

La lógica de reintentos funciona así:

1. El master recibe un `TaskResult` con `success = false`.
2. Incrementa contadores de fallos para el job y actualiza métricas globales.
3. Si `attempt <MAX_TASK_ATTEMPTS`, genera una nueva tarea con el mismo contenido pero `attempt + 1`.
4. Inserta esta tarea en la cola de *failed tasks*, que es consumida periódicamente por la tarea de `retry_failed_tasks`.
5. Si se alcanzó el máximo de intentos, el job se marca como `Failed`.

El script `test_failure_retries.sh` (incluido en los `.sh`) lanza un worker con `FAIL_PROBABILITY = 50` y comprueba que, a pesar de los fallos, el job puede llegar a completarse gracias a los reintentos.

## 9.3. Replanificación por worker caído

Cuando un worker deja de enviar heartbeats:

- `monitor_workers` lo marca `Down`.
- El master identifica las tareas que estaban `Running` en ese worker y las trata como fallidas por worker caído.
- Si los intentos lo permiten, reinyecta esas tareas en la cola de pendientes para que otros workers las ejecuten.

El script `test_fault_tolerance.sh` incluye dos tests: uno de reintentos por fallo de tarea y otro donde se mata explícitamente un worker en mitad de la ejecución para comprobar que el job igual termina gracias a la replanificación.

## 9.4. Métricas de fallos

El endpoint `/api/v1/metrics/failures` concentra:

- Fallos por job (número de tareas fallidas).
- Estado de cada worker (Up/Down).
- Número de workers actualmente down.

Este endpoint es consultado en varios scripts después de inducir fallos, para verificar que las métricas coinciden con lo esperado (por ejemplo, que hay `job_failures >0` cuando se activan fallos simulados).

# 10. Persistencia de estado de jobs

## 10.1. Estructuras persistidas

La persistencia está encapsulada en `PersistedState` y `PersistedJob`:

- `PersistedState` contiene:
  - Un mapa `jobs: HashMap<String, PersistedJob>`.
  - Un `last_saved: u64` con el timestamp UNIX del último guardado.
- `PersistedJob` resume un job:
  - `id, name, status` como cadena.
  - `progress` y `parallelism`.
  - `created_at` y `updated_at`, también como timestamps UNIX.

Estos datos se serializan a JSON y se guardan en:

- Directorio: `/tmp/minispark/state`.
- Archivo principal: `jobs.json`.

## 10.2. Política de guardado incremental

Para no escribir en disco de forma excesiva:

- Cada vez que se actualiza un job, se llama a `PersistenceManager::save_job`.
- Este método actualiza el job en la estructura in-memory, pero sólo llama a `save()` si la diferencia entre el tiempo actual y `last_saved` es mayor o igual a `SAVE_INTERVAL_SECS`.
- La función `save()` se encarga de crear el directorio, serializar el estado de forma bonita (`to_string_pretty`) y escribir el JSON.

Además, existe un método `flush()` para forzar guardado inmediato, utilizado por tareas periódicas o scripts de prueba.

## 10.3. Carga de estado y diagnóstico

Al arrancar, el master:

1. Llama a `PersistedState::load()`.
2. Si el archivo no existe, inicializa un estado vacío y escribe en logs que no hay estado previo disponible.
3. Si sí existe, intenta parsear el JSON; si falla la deserialización, escribe un error y también arranca de estado vacío.
4. Al crear `PersistenceManager`, se llama a `get_incomplete_jobs()`, que devuelve los jobs cuyo estado persistido era `Running` o `Accepted`. Esta lista se imprime en logs para diagnóstico.

El endpoint `/api/v1/state` permite inspeccionar el estado persistido actual, facilitando depuración o demostraciones sobre cómo se guardan los jobs entre ejecuciones.

# 11. Métricas y observabilidad

## 11.1. Métricas del worker

El `MetricsCollector` registra:

- **Contadores de tareas**: total ejecutadas, total exitosas, total fallidas, tareas activas.
- **Latencias** de tareas: se mantiene una ventana de las últimas `LATENCY_WINDOW_SIZE` latencias en milisegundos, a partir de la cual se calcula:
  - Latencia promedio (`avg_latency_ms`).
  - Percentil 50 (mediana).
  - Percentil 99.
- **Throughput**: número de registros procesados por segundo, calculado a partir del aumento en `records_processed` por unidad de tiempo.
- **Métricas de sistema**: CPU, memoria usada y total, porcentaje de memoria.

Estas métricas se combinan con las estadísticas de cache (hits, misses, memoria ocupada) para formar un objeto `WorkerMetrics`, devuelto por `get_metrics()` y expuesto a través del endpoint `/metrics`.

## 11.2. Métricas del master

El master mantiene y expone:

- Estadísticas globales de jobs: cuántos están en cada estado (Accepted, Running, Succeeded, Failed).
- Cola de tareas pendientes y en ejecución.
- Carga por worker, combinando heartbeats y contadores internos.
- Métricas de fallos, reintentos y workers down.

Los scripts `test_metrics.sh` y `test_metrics_client.sh` activan la mayoría de los endpoints de métricas y verifican que:

- Los endpoints responden con JSON bien formado.
- Los valores cambian de forma razonable antes, durante y después de ejecutar un job.

## 11.3. Logging estructurado

Aunque no es un sistema de logging de producción, se utiliza una estructura `Logger` que permite crear mensajes con campos clave–valor (por ejemplo, `worker_id`, `job_id`, `error`). Esto facilita leer logs complejos al depurar, especialmente durante tests de fallos y replanificación.

# 12. Scripts de prueba y pipeline de validación

## 12.1. Pruebas unitarias

El script `run_tests.sh` ejecuta las pruebas unitarias de todo el workspace (`cargo test -workspace`) y recopila el output en un archivo temporal. Después cuenta cuántas pruebas pasaron y fallaron, imprime un resumen y falla si hay alguna prueba fallida.

## 12.2. Pruebas E2E single-node

El script de test single-node:

1. Limpia el estado de `/tmp/minispark` o `C:/tmp/minispark`.
2. Compila el proyecto en modo release.
3. Levanta un master y un worker único con un número fijo de hilos.
4. Envía un job de WordCount sobre el archivo `data/input.csv`.
5. Espera a que el job termine y muestra los resultados con `client results`.

Esta prueba verifica el ciclo completo master–worker–client en la configuración más simple posible.

## 12.3. Pruebas multinodo y join

Otros scripts E2E (`test_e2e_multinode_join.sh`, `test_e2e_multinode_wordcount.sh`) lanzan un master y tres workers con diferentes configuraciones de hilos. Se envían jobs de join y wordcount con paralelismo alto para:

- Asegurar que el scheduler realmente distribuye trabajo entre múltiples workers.
- Validar operadores más complejos, como join con shuffles.
- Extraer métricas del master y comprobar que reflejan la realidad del clúster.

## 12.4. Pruebas de tolerancia a fallos

Los scripts dedicados a fallos (`test_fault_tolerance.sh`, `docker-fault-test.sh`, `test_failure_retries.`
configuran el sistema en escenarios adversos:

- Workers con `FAIL_PROBABILITY` alto.
- Workers que se matan explícitamente durante la ejecución.
- Jobs con múltiple particiones para aumentar la probabilidad de fallo.

Estas pruebas demuestran la capacidad del sistema para completar jobs a pesar de fallos, mediante reintentos y replanificación.

## 12.5. Pruebas de cache, persistencia y métricas

El script `test_cache.sh` combina:

- Ejecución de un job con una cantidad de datos suficiente para forzar spill.
- Inspección de archivos en `/tmp/minispark/spill`.
- Verificación del archivo de estado `jobs.json`.
- Reinicio del master y consulta del endpoint `/api/v1/state` para confirmar que el estado persistido se carga correctamente.

El script `test_metrics.sh` se centra en:

- Consultar métricas del sistema antes y después de un job.
- Consultar métricas específicas de job, de etapas y de workers.
- Mostrar un resumen final de todos los endpoints de métricas implementados.

### 12.6. Benchmarks y demos Docker

Por último, scripts como `docker-demo.sh` y `benchmark_1m.sh` ejecutan:

- Creación de datasets sintéticos (100 líneas, 1000 líneas, 1 000 000 líneas).
- Compilación y construcción de imágenes Docker.
- Levantado de clúster en Docker con uno o varios workers.
- Ejecución de jobs y medición del tiempo total en milisegundos.
- Recolección de métricas del master y consolidación en archivos JSON en `results/`.

Esto proporciona un *benchmark oficial* para Mini-Spark y una demo reproducible del sistema en un entorno containerizado.

## 13. Configuración y parámetros de despliegue

El comportamiento del sistema se ajusta mediante variables de entorno y rutas estándar, lo cual simplifica la experimentación:

- `MASTER_URL`: URL base del master para los workers y el cliente (por defecto `http://127.0.0.1:8080`).
- `WORKER_PORT`: puerto base de cada worker (9000, 9010, 9020, etc.).
- `WORKER_THREADS`: número de ejecutores lógicos por worker (controla paralelismo interno).
- `CACHE_MAX_MB`: límite de memoria de cache en MB.
- `FAIL_PROBABILITY`: probabilidad (0–100) de fallo simulado por cada tarea.

En Windows, los scripts adaptan rutas a `C:/tmp/minispark`, mientras que en Linux se usa `/tmp/minispark`. El código Rust no está acoplado a Windows o Linux, salvo por la lectura de `/proc` para métricas del sistema, que naturalmente es específica de entornos tipo Unix.

## 14. Conclusiones

Mini-Spark integra en un solo proyecto muchos conceptos clave de sistemas operativos y sistemas distribuidos:

- Uso combinado de procesos y hilos (o tasks asíncronas) para lograr paralelismo y concurrencia.
- Planificación de tareas entre múltiples workers usando un scheduler simple pero consciente de la carga.
- Gestión de memoria explícita mediante una cache con límite y política de spill a disco.
- Manejo de fallos a nivel de tarea y worker con reintentos y replanificación.
- Persistencia de estado y métricas exhaustivas para hacer el sistema observable y depurable.

La implementación en Rust, la estructura clara de módulos y la batería extensa de scripts de prueba convierten a Mini-Spark en una plataforma pequeña pero real para experimentar con principios de sistemas operativos y de computación distribuida, cerrando el ciclo desde el diseño conceptual hasta la validación práctica en escenarios con carga, fallos y despliegue multinodo.