Informe Proyecto 1: GridMR

ST0263 Tópicos Especiales en Telemática

Juan Miguel Muñoz García

Juan David Zapata Moncada

Universidad EAFIT, Medellín

Escuela de Ciencias Aplicadas e ingenieria

Profesor: Álvaro Enrique Ospina Sanjuan

Medellín, Colombia

16 de septiembre de 2025

1. Descripción del servicio y problema abordado

El sistema GridMR es una forma de procesamiento distribuido que permite a un cliente enviar tareas (jobs) para ser resueltas sobre una red de nodos heterogéneos y débilmente acoplados. En este caso, el servicio se basa en el paradigma **MapReduce**, que permite separar el procesamiento de las tareas, esto es, dividiendo los problemas complejos en sub-tareas (Map) y luego, consolidar los resultados (Reduce).

El problema que se aborda con este proyecto es la dificultad que se tiene a la hora de procesar grandes volúmenes de datos en un único nodo o equipo. Cuando este problema se hace presente, la mejor opción es optar por sistemas de procesamiento y/o almacenamiento distribuido, de esta forma se garantiza la resolución de tareas con alta demanda computacional.

GridMR propone una solución modular y flexible en la que cada nodo de la red puede ayudar en el procesamiento de datos a través de APIS de comunicación. Lo anterior permite que el cliente acceda a un servicio de este estilo sin necesidad de conocer la infraestructura que se encuentra detrás, ocupándose solo de enviar una tarea a ser procesada.

Así pues, con este servicio abordamos el reto de procesamiento masivos de datos en sistemas distribuidos garantizando balance de carga y recolección de resultados.

2. Arquitectura del sistema

El sistema GridMR se diseñó bajó una arquitectura Master-Workers, que corresponde a una variante de arquitectura Cliente-Servidor. En esta arquitectura encontramos tres tipos principales de Nodos:

- **Cliente:** Este componente se encarga de enviar el job con sus respetivas funciones Map() y Reduce(), así como los parámetros de ejecución (tamaño de Split, número de reducers, etc.). El cliente envía las solicitudes al Maestro y recibe los resultados procesados.

  Con el fin de mejorar la experiencia de usuario, se desarrolló una interfaz grafica para este componente con **Streamlit.** Para ejecutar este componente, véase el README del repositorio de github.

- **Maestro:** Es el encargado de la coordinación general del servicio. Su función principal es dividir el trabajo y repartirlo entre los distintos Map workers, así como consolidar los resultados finales entregados por los Reduce Workers y retornarlos al cliente. Es el componente principal del sistema.

- **Workers:** Son los nodos encargados de procesar la data que se envía desde el cliente. Están empaquetados en contenedores Docker y cada uno expone una API para recibir datos y devolver resultados. En este caso, los dividimos en dos tipos:

- **Map Workers:** Reciben fragmentos de datos desde el maestro, ejecutan la función map() y producen resultados intermedios.
- **Reduce Workers:** Reciben los resultados intermedios, ejecutan la función reduce() y devuelven resultados finales al maestro.

La comunicación entre cada componente se realiza a través de REST sobre HTTP con FastAPI, de esta forma cada nodo puede encontrarse distribuido en diferentes puntos de la red.

Este diseño permite que el sistema sea escalable, ya que se pueden añadir más workers según la demanda. Además, al estar empaquetados dentro de contenedores Docker, se hacen muy portables y flexibles.

Para revisar los diagramas correspondientes a la arquitectura, véase la carpeta 'diagramas' del proyecto, ubicada en esta misma carpeta.

3. Especificación de protocolos y APIs

La comunicación en el sistema GridMR se implementa mediante REST sobre HTTP, intercambiando mensajes en formato JSON. Esto permite que los nodos cliente, maestro y trabajadores se comuniquen de forma desacoplada y sencilla. A continuación, se hace una descripción de los endpoints definidos:

1. Cliente ↔ Maestro

POST /submit_job
Envía un nuevo trabajo al maestro.
Request (JSON):

```
{
  "data": "texto de entrada o ubicación de archivo",
  "split_size": 100,
  "num_reducers": 2,
  "map_function": "word_count_map",
  "reduce_function": "word_count_reduce"
}
```

Response (JSON):

```
{
  "job_id": "uuid-del-trabajo",
  "status": "submitted"
}
```

GET /job_status/{job_id}
Consulta el estado de un trabajo específico.
Response (JSON):

```
{
  "job_id": "uuid-del-trabajo",
```

```
  "status": "running"
}
```

GET /job_result/{job_id}

Obtiene el resultado final de un trabajo terminado.

Response (JSON):

```
{
  "job_id": "uuid-del-trabajo",
  "result": {
    "palabra1": 5,
    "palabra2": 3
  }
}
```

---

2. Maestro ↔ Workers

POST /map_task

Asigna a un worker de tipo Map un fragmento de datos para procesar.

Request (JSON):

```
{
  "job_id": "uuid-del-trabajo",
  "split_id": 0,
  "data": "fragmento de texto"
}
```

Response (JSON):

```
{
  "split_id": 0,
  "intermediate_results": {
    "palabra1": [1,1],
    "palabra2": [1]
  }
}
```

POST /reduce_task

Envía a un worker de tipo Reduce los resultados intermedios recolectados para consolidarlos.

Request (JSON):

```
{
  "job_id": "uuid-del-trabajo",
  "reduce_id": 1,
  "intermediate_data": {
    "palabra1": [1,1],
    "palabra2": [1]
  }
}
```

Response (JSON):


```
{
 "reduce_id": 1,
 "final_result": {
  "palabra1": 2,
  "palabra2": 1
 }
}
```


Flujo básico de comunicación:

1. El cliente envía un Job al maestro (/submit_job)
2. El maestro divide el texto en splits y asigna cada uno a un worker Map (/map_task)
3. Los workers Map responden con resultados intermedios.
4. El maestro agrupa los resultados intermedios y los envía a workers Reduce (/reduce_task).
5. Los workers Reduce responden con resultados finales.
6. El maestro reúne la información y la pone disponible para el cliente en (/job_result/{job_id}), mientras que (/job_status/{job_id}) permite monitorear la ejecución.

4. Algoritmo de planificación

El sistema de planificación para este proyecto se divide en varias fases y mecanismos:

1. División de datos:
   El nodo Maestro divide los datos de entrada en fragmentos o splits, según el parámetro definido por el cliente (split_size).

2. Asignación de tareas Map:
   Para asignar cada split a un worker Map se utiliza un planificador con balanceo de carga:
   - El maestro consulta el endpoint /status de cada worker para conocer el número de tareas en progreso.
   - Se selecciona el worker menos cargado mediante la función get_least_loaded_worker().
   - En caso de fallo en la comunicación, se implementa un mecanismo de reintentos (send_with_retry) y reubicación del split en otro worker disponible.

   Este enfoque corresponde a un algoritmo de **Least Connection Scheduling**, que asegura que ningún worker quede sobrecargado.

3. Recolección de resultados intermedios
   Los resultados generados por cada worker Map se devuelven al maestro, quien los almacena en un diccionario para asignarlos a los reduce workers.

4. Asignación tareas Reduce
   Esta fase se planifica usando un Hash partitioner, de esta forma se garantiza que todas las ocurrencias de una misma palabra vayan al mismo worker. Así:
   - Cada clave producida por la fase Map se asigna a un reduce_worker calculado como hash(clave) % num_reducers.
   - Los datos asignados se envían a workers de tipo Reduce a través del endpoint /reduce_task.

5. Consolidación de resultados
   Finalmente, los resultados parciales de cada Reduce se integran en un resultado final, almacenado en memoria por el maestro y disponible en el endpoint /job_result/{job_id}.

5. Descripción de entorno de ejecución

Para la simulación de este servicio se utilizó Docker Compose, que define los paquetes necesarios:

- **Mestro:** Contenedor que ejecuta la aplicación FastAPI encargada de la planificación y asignación de tareas. Expone el puerto 8000.
- **Maps (1-4):** Contenedores de tipo Map Worker, cada uno ejecutando un servicio con su propio puerto (8001, 8003, 8005, 8007). Reciben splits de datos y ejecutan la función map().
- **Reducers (1-**2): Contenedores de tipo Reduce Worker, expuestos en los puertos 8002 y 8004, encargados de recibir resultados intermedios desde el Maestro y aplicar la función reduce() sobre esos resultados.

Adicional, el archivo docker-compose.yml describe la red de servicios y define las imágenes para cada tipo de nodo (./maestro, ./workers). Así como los comandos de arranque mediante uvicorn para exponer la API en cada contenedor. Es importante aclarar que tanto Maestro como Workers tienen su propio Dockerfile, en el cual se encuentra las dependencias de cada uno.

Para la ejecución local del proyecto, siga los pasos indicados en el README del repositorio.

6. Análisis de resultados y rendimiento

Para evaluar el rendimiento de GridMR, se realizaron pruebas con diferentes configuraciones de tamaño de split, número de reducers y volumen de datos de entrada.

Principalmente, se observó que, con volúmenes de datos pequeños, el tiempo de computo era muy bajo y constante. Por otro lado, al incrementar el número de Map workers, se pudo evidenciar que los tiempos de procesamiento disminuían. Esto mismo ocurría si se aumentaba

la cantidad de splits manteniendo una cantidad proporcional de Map Workers. En cuanto al número de reducers, no presentaron muchas dificultades en lo que a escalabilidad se refiere, incluso se logra evidenciar como el maestro asigna adecuadamente los resultados intermedios a estos nodos.

Continuando con lo mencionado al final del párrafo, se observa que el sistema puede escalar horizontalmente, basta con añadir nuevos servicios desde el archivo de docker-compose.yml para incrementar la capacidad de procesamiento.

En cuanto a la autoevaluación, se evidencia que si bien se logra el objetivo principal del proyecto, que es distribuir el procesamiento de los datos, también se debe tener en cuenta que quedan algunas funcionalidades incompletas, especialmente en el módulo de la gestión de datos. En general, se podría decir que se completó el 80% del proyecto.