Procesamiento de Grandes Volúmenes de Datos

Conferencia 5: Consistencia y la Gestión de Recursos

Deborah Famadas Rodríguez

Universidad de La Habana

6 de octubre de 2025

Recordemos...

Los sistemas distribuidos enfrentan tres grandes retos:

Retos Clave

Sincronización de estado — Replicación y particionamiento — Tolerancia a fallos

El **Teorema CAP** nos obliga a elegir entre **Consistencia** y **Disponibilidad** ante una Partición de red.

Linearizabilidad

Un sistema es **linearizable** si todas las operaciones parecen ocurrir en un único instante.

Linearizabilidad

Un sistema es **linearizable** si todas las operaciones parecen ocurrir en un único instante. *Si A termina antes de que B comience, B debe ver el efecto de A.*

Consistencia Secuencial

Definición

Todas las operaciones se ejecutan en un orden secuencial global, aunque no respete el tiempo real.

Consistencia Secuencial

Definición

Todas las operaciones se ejecutan en un orden secuencial global, aunque no respete el tiempo real.

Clave

Todos los nodos ven el mismo historial, pero ese historial puede no coincidir con el reloj de pared.

Comparando Modelos

Linearizabilidad

- Modelo más fuerte e intuitivo.
- Respeta el tiempo real.
- Mayor costo en latencia.

Consistencia Secuencial

- Orden global consistente.
- Ignora el tiempo real.
- Simplifica la programación.

El Problema Fundamental

Ya conocemos frameworks como **MapReduce** y **Spark**.

El Problema Fundamental

Ya conocemos frameworks como **MapReduce** y **Spark**. Pero surge la pregunta:

¿Quién decide cómo se usan los 1000 nodos del clúster?

¿Quién asigna CPU y memoria? ¿Quién evita la interferencia entre trabajos?

El Problema Fundamental

Ya conocemos frameworks como **MapReduce** y **Spark**. Pero surge la pregunta:

¿Quién decide cómo se usan los 1000 nodos del clúster?

¿Quién asigna CPU y memoria? ¿Quién evita la interferencia entre trabajos?

La respuesta: **un Gestor de Recursos**.

Objetivos del Planificador

Un buen planificador busca equilibrar:

Objetivos del Planificador

Un buen planificador busca equilibrar:

Criterios

Rendimiento (Throughput) — Latencia — Equidad — Utilización

Objetivos del Planificador

Un buen planificador busca equilibrar:

Criterios

Rendimiento (Throughput) — Latencia — Equidad — Utilización

No existe un algoritmo universal

FIFO, Round-Robin o Prioridades se eligen según el contexto.

Planificación FIFO

- Primer trabajo en llegar, primer trabajo en ejecutarse.
- Simple y fácil de implementar.
- Bajo costo de planificación.

FIFO procesa los trabajos en orden de llegada. Es sencillo pero puede provocar el *efecto convoy*, donde un trabajo largo retrasa a todos los demás.

Planificación Round-Robin

- Cada trabajo recibe un *quantum* de tiempo o recursos.
- Los trabajos rotan en turnos.
- Evita bloqueos por trabajos largos.

Round-Robin funciona como un carrusel. Cada trabajo avanza en rondas. Elegir el tamaño adecuado del *quantum* es importante para evitar sobrecarga o demoras excesivas.

Planificación por Prioridades

- Los trabajos tienen niveles de prioridad.
- Se atienden primero los más críticos.
- Riesgo: trabajos de baja prioridad pueden quedar relegados (*starvation*).

Este esquema es flexible y permite dar prioridad a trabajos urgentes. Se suele usar *aging* para que los trabajos de baja prioridad no queden olvidados.

La Era del JobTracker

Hadoop v1 fusionaba:

- Gestión de recursos.
- Gestión de aplicaciones MapReduce.

La Era del JobTracker

Hadoop v1 fusionaba:

- Gestión de recursos.
- Gestión de aplicaciones MapReduce.

Problemas

Cuello de botella — SPOF — Acoplado a MapReduce — Desperdicio de recursos

La Gran Idea de YARN

Separación de responsabilidades

- Gestión de recursos: global y agnóstica.
- Lógica de aplicación: delegada a cada framework.

La Gran Idea de YARN

Separación de responsabilidades

- Gestión de recursos: global y agnóstica.
- Lógica de aplicación: delegada a cada framework.

Resultado: YARN actúa como un **Sistema Operativo del Clúster**.

¿Qué es YARN?

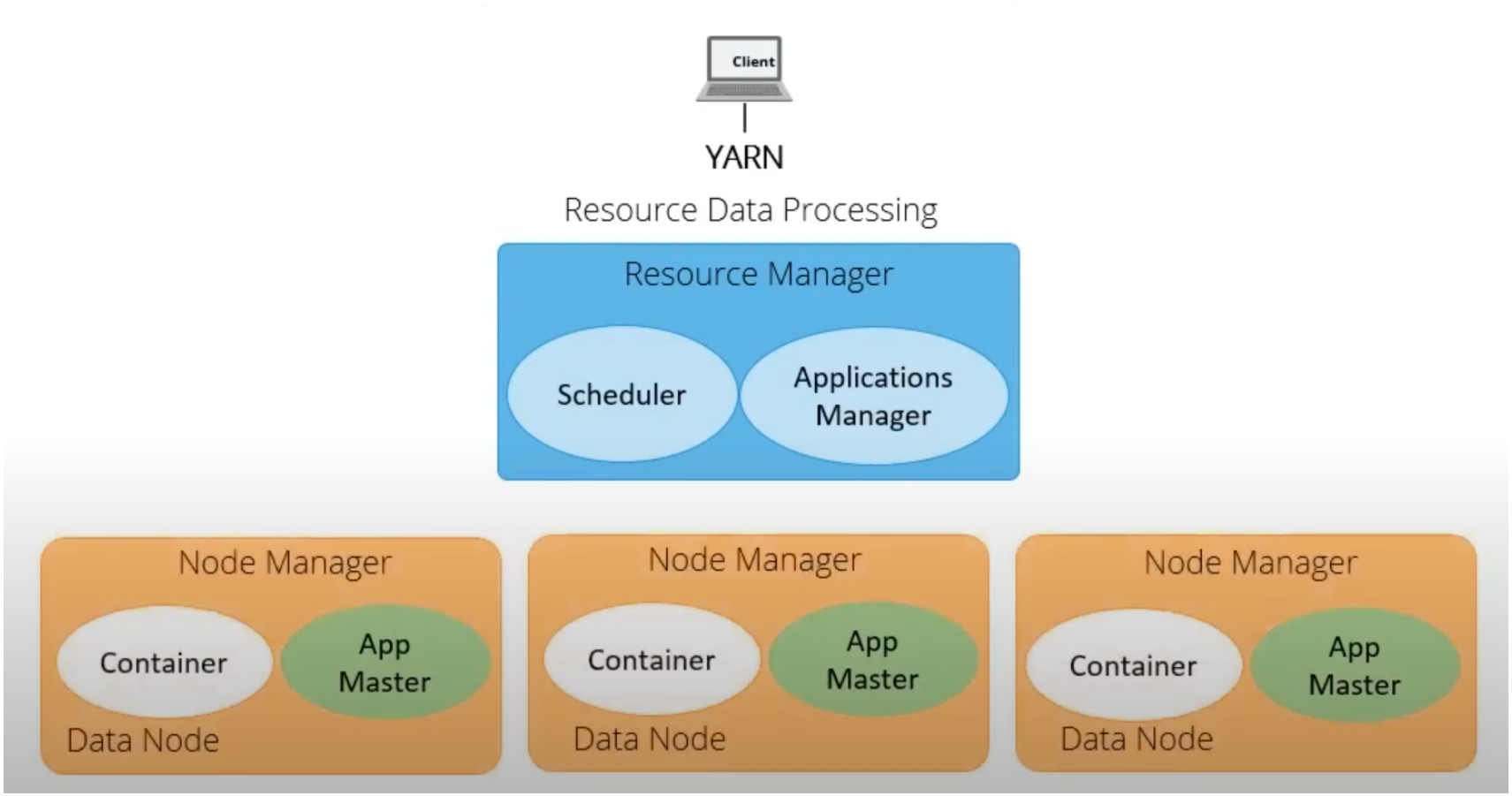
YARN es el acrónimo de *Yet Another Resource Negotiator* (Otro Negociador de Recursos Más).

- Es un gestor de recursos creado al separar el motor de procesamiento y la gestión de **MapReduce**.
- Se encarga de monitorear y administrar cargas de trabajo.
- Mantiene un entorno multi-inquilino (multi-tenant).
- Gestiona las características de alta disponibilidad de Hadoop.
- Implementa controles de seguridad.

Ventajas del Enfoque de Clúster Único

- **Mayor utilización del clúster:** Los recursos no utilizados por un framework pueden ser consumidos por otro.
- **Menores costos operativos:** Solo se necesita gestionar un único clúster "que lo hace todo".
- **Reducción del movimiento de datos:** No hay necesidad de mover datos entre diferentes clústeres.

Arquitectura de Yarn



Componentes de YARN

- **ResourceManager:** Maestro global, contiene el planificador.
- **NodeManager:** Agente en cada nodo, gestiona contenedores.
- **ApplicationMaster:** Uno por aplicación, negocia recursos.
- **Contenedor:** Unidad básica de asignación.

1. Gestor de Recursos (Resource Manager - RM)

Es el servidor maestro, y generalmente hay uno por clúster.

- Conoce la ubicación de los nodos de datos y cuántos recursos tienen, lo que se conoce como **conciencia de rack (rack awareness)**.

- Ejecuta varios servicios, siendo el más importante el **Planificador de Recursos (Resource Scheduler)**, que decide cómo asignar los recursos.

Análisis Detallado del Gestor de Recursos (RM)

El RM media los recursos disponibles en el clúster entre las aplicaciones competidoras, con el objetivo de maximizar la utilización del clúster.

Componentes Principales del RM:

- **Planificador (Scheduler):**
  - Asigna recursos a las aplicaciones según restricciones (capacidad, colas, etc.).
  - No monitorea el estado de la aplicación ni reinicia tareas si fallan.
  - Funciona basado en la noción de un **contenedor de recursos** (memoria, CPU, disco, red).
- **Gestor de Aplicaciones (Applications Manager):**
  - Acepta envíos de trabajos.
  - Negocia el primer contenedor para ejecutar el Application Master.
  - Reinicia el contenedor del Application Master en caso de fallo.

2. Maestro de Aplicación (Application Master - AM)

Es un proceso específico del framework que negocia recursos para **una sola aplicación**.

- Se ejecuta en el primer contenedor asignado para la aplicación.
- Solicita recursos al Resource Manager y luego trabaja con los contenedores proporcionados por los Node Managers.

Responsabilidades:

- Gestiona el ciclo de vida de la aplicación.
- Realiza ajustes dinámicos en el consumo de recursos.
- Maneja el flujo de ejecución, fallos y proporciona estado y métricas.
- No se considera confiable y no se ejecuta como un servicio de confianza.
- Cada aplicación tiene su propia instancia de un Application Master.

3. Gestores de Nodos (Node Managers - NM)

Son los esclavos de la infraestructura; puede haber muchos en un clúster.

- Al iniciarse, se anuncian al RM y periódicamente le envían una señal de "latido"(heartbeat).
- Cada NM ofrece recursos al clúster, definidos por la memoria y el número de **vCores** (núcleos virtuales).
- Un **contenedor (container)** es una fracción de la capacidad de un NM y es utilizado por el cliente para ejecutar un programa.
- Cada NM recibe instrucciones del RM y gestiona los contenedores en un único nodo.

Responsabilidades del Gestor de Nodos (NM)

Cuando se asigna un contenedor, el NM configura su entorno, incluyendo restricciones y dependencias (archivos, ejecutables).

Responsabilidades Clave:

- Monitorea la salud del nodo e informa al RM sobre problemas de hardware o software.
- Ofrece servicios a los contenedores, como la agregación de logs.
- Gestiona el ciclo de vida del contenedor, sus dependencias y el uso de recursos.
- Informa el estado del nodo y del contenedor al Resource Manager.

Flujo de Spark en YARN

1. Cliente solicita aplicación.
2. RM lanza un contenedor inicial.
3. AM negocia recursos.
4. RM asigna contenedores.
5. AM lanza ejecutores.
6. Ejecución con supervisión de YARN.

Planificadores en YARN

Capacity Scheduler

Garantiza porcentajes fijos de capacidad por cola. Útil en entornos corporativos.

Planificadores en YARN

Capacity Scheduler

Garantiza porcentajes fijos de capacidad por cola. Útil en entornos corporativos.

Fair Scheduler

Reparte recursos dinámicamente. Útil en entornos multiusuario.

¿Qué es Kubernetes?

Kubernetes (comúnmente abreviado como **K8s**) es una plataforma *open-source* para automatizar el despliegue, el escalado y la gestión de aplicaciones en contenedores.

- Originalmente diseñado por **Google**, ahora es mantenido por la **Cloud Native Computing Foundation (CNCF)**.
- Permite gestionar clústeres de hosts como un único recurso computacional.
- Su objetivo es proporcionar una "plataforma para automatizar el despliegue, escalado y operaciones de contenedores de aplicaciones a través de clústeres de hosts".

El Problema que Resuelve

Gestionar aplicaciones en contenedores a escala es complejo.
Antes de la orquestación:

- Despliegues y actualizaciones manuales propensos a errores.
- Dificultad para escalar aplicaciones según la demanda.
- Sin mecanismos automáticos de recuperación ante fallos de un contenedor o un nodo.
- Complejidad en el descubrimiento de servicios y el balanceo de carga.
- Inconsistencias entre los entornos de desarrollo, pruebas y producción.

→ **Kubernetes automatiza y soluciona estos desafíos.**

Conceptos Fundamentales

- **Contenedor (Container):** Paquete ligero y ejecutable que incluye todo lo necesario para correr una aplicación: código, librerías y dependencias. Comúnmente Docker.
- **Pod:** La unidad de despliegue más pequeña en Kubernetes. Es un grupo de uno o más contenedores que comparten almacenamiento, red y especificaciones de ejecución.
- **Nodo (Node):** Una máquina de trabajo en un clúster, que puede ser una máquina virtual o física. Es donde se ejecutan los Pods.
- **Clúster (Cluster):** Un conjunto de Nodos agrupados. Está compuesto por un Plano de Control (Control Plane) y Nodos de trabajo (Worker Nodes).

Flujo de Trabajo Típico en Kubernetes

Kubernetes funciona bajo un modelo **declarativo**.

1. El usuario define el **estado deseado** de la aplicación en un archivo de manifiesto (generalmente en formato YAML).
2. Se envía este manifiesto al **API Server** usando una herramienta como ‘kubectl‘.
3. El **Plano de Control** (Scheduler, Controller Manager) trabaja para que el estado actual del clúster coincida con el estado deseado.
4. El **Scheduler** asigna los Pods a los Nodos.
5. El **Kubelet** de cada Nodo se asegura de crear y mantener los contenedores necesarios.
6. Los controladores monitorean continuamente el clúster para corregir cualquier desviación del estado deseado (ej. si un Pod falla, lo reinician).

¿Qué es Apache Mesos?

Apache Mesos es un proyecto *open-source* para gestionar clústeres de computadoras. Fue diseñado para proporcionar aislamiento y compartición de recursos eficiente entre aplicaciones distribuidas.

- Se originó en el **RAD Lab** de la **UC Berkeley**.
- Su núcleo abstrae CPU, memoria, almacenamiento y otros recursos computacionales de las máquinas (físicas o virtuales).
- Permite que sistemas de *workloads* complejos y elásticos sean fáciles de construir y ejecutar.
- La analogía central es que **Mesos es un "kernel"para tu centro de datos**.

La Filosofía: El Kernel del Centro de Datos

Así como el kernel de un sistema operativo (como Linux) abstrae el hardware de una máquina para las aplicaciones, Mesos abstrae los recursos de un conjunto de máquinas para los frameworks.

Kernel de SO Tradicional

- Abstrae CPU, RAM, Disco
- Proporciona APIs (p.ej., POSIX)
- Permite a múltiples procesos compartir recursos de forma segura.

Kernel Distribuido (Mesos)

- Abstrae un pool de máquinas
- Proporciona APIs para frameworks
- Permite a múltiples sistemas distribuidos (frameworks) compartir un clúster.

$\rightarrow$

El objetivo es la máxima utilización de recursos en un clúster compartido.

Componentes Principales de la Arquitectura

- **Mesos Master:** Coordina el clúster. Es responsable de ofrecer los recursos disponibles a los frameworks. Se utiliza **ZooKeeper** para la elección de un líder y lograr alta disponibilidad.

- **Mesos Agent (Agente):** (Antes conocido como *Slave*) Se ejecuta en cada nodo del clúster. Es responsable de ejecutar las tareas y reportar los recursos disponibles al Master (ej: 4 CPU, 8 GB RAM).

- **Framework:** Es una aplicación que corre sobre Mesos. Cada framework consta de dos partes:
  - Un **Scheduler** (Planificador) que se registra en el Master para recibir ofertas de recursos.
  - Un **Executor** (Ejecutor), un proceso que se lanza en los nodos Agente para ejecutar las tareas del framework.

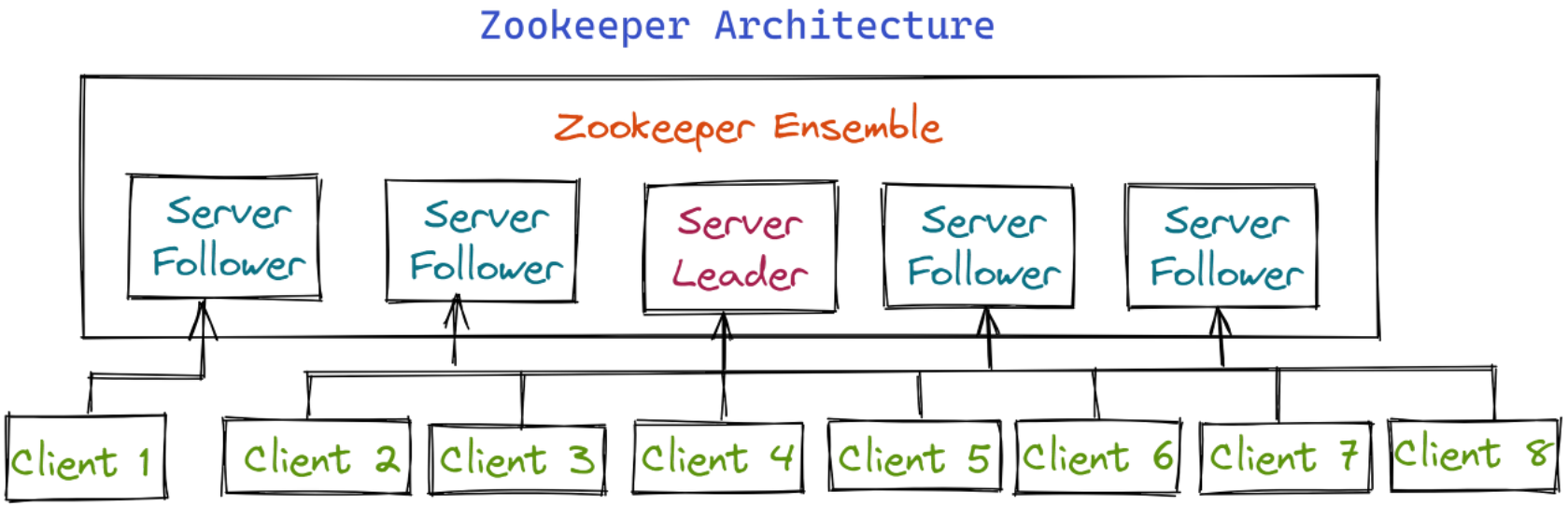
Apache ZooKeeper: Coordinación para Sistemas Distribuidos

¿Qué es ZooKeeper?

Es un servicio centralizado de código abierto que proporciona un **primitivo fiable** para construir aplicaciones distribuidas. Su función es eliminar la complejidad de la sincronización y la coordinación.

- Mantiene información de configuración, nombramiento y estado.
- Es utilizado por sistemas complejos como Hadoop, Kafka y Mesos para tareas críticas como la elección de un líder o el descubrimiento de nodos.
- **Analogía Principal:** Funciona como un *sistema de archivos distribuido* muy simple y altamente disponible.

Zookeeper Arquitecture



Tendencias Futuras

1. **Serverless:** Abstracción total de la infraestructura.
2. **IA/ML:** Planificadores inteligentes con Aprendizaje por Refuerzo.

Tendencias Futuras

1. **Serverless:** Abstracción total de la infraestructura.
2. **IA/ML:** Planificadores inteligentes con Aprendizaje por Refuerzo.

Objetivo Final

Sistemas autónomos, eficientes y fáciles de usar.

Computación Serverless

- Abstracción total de la infraestructura.
- El desarrollador escribe funciones, no gestiona servidores.
- Escalado automático y pago por uso.

El modelo Serverless simplifica el despliegue: el usuario no administra máquinas, solo funciones. Se escala automáticamente y se factura según el consumo real.

Serverless + IA

- Planificadores inteligentes con aprendizaje por refuerzo.
- Sistemas autónomos, eficientes y fáciles de usar.

El futuro combina Serverless con IA, creando planificadores que aprenden y optimizan el uso de recursos sin intervención humana.

Evolución de la Gestión de Recursos

1. Monolíticos (Hadoop 1.x).
2. Frameworks dedicados (YARN).
3. Orquestadores universales (Kubernetes).
4. Sistemas autónomos (Serverless + IA).

¿Preguntas?