Resource Management
Mesos and YARN

Amir H. Payberah
amir@sics.se

KTH Royal Institute of Technology
Motivation

- Rapid innovation in cloud computing.
- No single framework optimal for all applications.
Motivation

- Rapid innovation in cloud computing.
- No single framework optimal for all applications.
- Running each framework on its dedicated cluster:
  - Expensive
  - Hard to share data
Proposed Solution

- Running multiple frameworks on a single cluster.

- Maximize utilization and share data between frameworks.

- Two resource management systems:
  - Mesos
  - YARN
Mesos
Mesos

A common resource sharing layer, over which diverse frameworks can run
Mesos

A common resource sharing layer, over which diverse frameworks can run
Mesos Goals

- High utilization of resources
- Support diverse frameworks (current and future)
- Scalability to 10,000’s of nodes
- Reliability in face of failures
A framework (e.g., Hadoop, Spark) manages and runs one or more jobs.
Computation Model

- A framework (e.g., Hadoop, Spark) manages and runs one or more jobs.
- A job consists of one or more tasks.
Computation Model

- A **framework** (e.g., Hadoop, Spark) manages and runs one or more **jobs**.
- A **job** consists of one or more **tasks**.
- A **task** (e.g., map, reduce) consists of one or more **processes** running on same machine.
Computation Model

- A **framework** (e.g., Hadoop, Spark) manages and runs one or more **jobs**.

- A **job** consists of one or more **tasks**.

- A **task** (e.g., map, reduce) consists of one or more **processes** running on same machine.
Mesos Design Elements

- Fine-grained sharing
- Resource offers
Fine-Grained Sharing

- Allocation at the level of tasks within a job.
- Improves utilization, latency, and data locality.

Coarse-grained sharing vs. Fine-grained sharing
Resource Offer

- Offer available resources to frameworks, let them pick which resources to use and which tasks to launch.

- Keeps Mesos simple, lets it support future frameworks.
How to schedule resource offering among frameworks?
Schedule Frameworks

- **Global** scheduler
- **Distributed** scheduler
Global Scheduler (1/2)

- **Job requirements**
  - Response time
  - Throughput
  - Availability

- **Job execution plan**
  - Task DAG
  - Inputs/outputs

- **Estimates**
  - Task duration
  - Input sizes
  - Transfer sizes
Global Scheduler (2/2)

▶ Advantages
  • Can achieve optimal schedule.

▶ Disadvantages
  • Complexity: hard to scale and ensure resilience.
  • Hard to anticipate future frameworks requirements.
  • Need to refactor existing frameworks.
Distributed Scheduler (1/3)

Organization policies -> Mesos Master -> Framework schedule

Resource availability -> Framework scheduler

Task schedule
Distributed Scheduler (2/3)

▶ Unit of allocation: resource offer
  • Vector of available resources on a node
  • For example, node1: \(\langle 1\text{CPU}, 1\text{GB} \rangle\), node2: \(\langle 4\text{CPU}, 16\text{GB} \rangle\)

▶ Master sends resource offers to frameworks.

▶ Frameworks select which offers to accept and which tasks to run.
Advantages

- **Simple**: easier to scale and make resilient.
- **Easy to port** existing frameworks, support new ones.

Disadvantages

- Distributed scheduling decision: **not optimal**.
Slaves continuously send status updates about resources to the Master.
Pluggable scheduler picks framework to send an offer to.
Framework scheduler selects resources and provides tasks.
▶ Framework executors launch tasks.
Question?

How to allocate resources of different types?
n users want to share a resource, e.g., CPU.  

- **Solution**: allocate each $\frac{1}{n}$ of the shared resource.
n users want to share a resource, e.g., CPU. 
- **Solution**: allocate each $\frac{1}{n}$ of the shared resource.

- Generalized by max-min fairness.
  - Handles if a user wants less than its fair share.
  - E.g., user 1 wants no more than 20%.
n users want to share a resource, e.g., CPU.

- **Solution**: allocate each \( \frac{1}{n} \) of the shared resource.

Generalized by **max-min fairness**.

- Handles if a user wants less than its fair share.
- E.g., user 1 wants no more than 20%.

Generalized by **weighted max-min fairness**.

- Give weights to users according to importance.
- E.g., user 1 gets weight 1, user 2 weight 2.
Max-Min Fairness

- 1 resource: CPU
- Total resources: 20 CPU
- User 1 has $x$ tasks and wants $\langle 1CPU \rangle$ per task
- User 2 has $y$ tasks and wants $\langle 2CPU \rangle$ per task
Max-Min Fairness

- 1 resource: CPU
- Total resources: 20 CPU
- User 1 has $x$ tasks and wants $\langle 1 CPU \rangle$ per task
- User 2 has $y$ tasks and wants $\langle 2 CPU \rangle$ per task

\[
\max(x, y) \text{ (maximize allocation)}
\]
Max-Min Fairness

- 1 resource: CPU
- Total resources: 20 CPU
- User 1 has $x$ tasks and wants $\langle 1CPU \rangle$ per task
- User 2 has $y$ tasks and wants $\langle 2CPU \rangle$ per task

$$\max(x, y) \text{ (maximize allocation)}$$
subject to
$$x + 2y \leq 20 \text{ (CPU constraint)}$$
$$x = 2y$$
Max-Min Fairness

- 1 resource: CPU
- Total resources: 20 CPU
- User 1 has \(x\) tasks and wants \(\langle 1CPU \rangle\) per task
- User 2 has \(y\) tasks and wants \(\langle 2CPU \rangle\) per task

\[
\max(x, y) \quad \text{(maximize allocation)}
\]

subject to
\[
x + 2y \leq 20 \quad \text{(CPU constraint)}
\]
\[
x = 2y
\]
so
\[
x = 10
\]
\[
y = 5
\]
Why is Fair Sharing Useful?

- **Proportional allocation**: user 1 gets weight 2, user 2 weight 1.

- **Priorities**: give user 1 weight 1000, user 2 weight 1.

- **Reservations**: ensure user 1 gets 10% of a resource, so give user 1 weight 10, sum weights $\leq 100$.

- **Isolation policy**: users cannot affect others beyond their fair share.
Properties of Max-Min Fairness

- **Share guarantee**
  - Each user can get at least $\frac{1}{n}$ of the resource.
  - But will get less if her demand is less.

- **Strategy proof**
  - Users are not better off by asking for more than they need.
  - Users have no reason to lie.
Properties of Max-Min Fairness

- **Share guarantee**
  - Each user can get at least $\frac{1}{n}$ of the resource.
  - But will get less if her demand is less.

- **Strategy proof**
  - Users are not better off by asking for more than they need.
  - Users have no reason to lie.

- **Max-Min fairness** is the only reasonable mechanism with these two properties.

- Widely used: OS, networking, datacenters, ...
Question?

When is Max-Min Fairness NOT Enough?
Question?
When is Max-Min Fairness NOT Enough?

Need to schedule multiple, heterogeneous resources, e.g., CPU, memory, etc.
Problem

- **Single resource example**
  - 1 resource: CPU
  - User 1 wants \(1\)CPU\) per task
  - User 2 wants \(2\)CPU\) per task
Problem

► **Single resource** example
  - 1 resource: CPU
  - User 1 wants $\langle 1CPU \rangle$ per task
  - User 2 wants $\langle 2CPU \rangle$ per task

► **Multi-resource** example
  - 2 resources: CPUs and mem
  - User 1 wants $\langle 1CPU, 4GB \rangle$ per task
  - User 2 wants $\langle 2CPU, 1GB \rangle$ per task
Problem

- **Single resource** example
  - 1 resource: CPU
  - User 1 wants \(1CPU\) per task
  - User 2 wants \(2CPU\) per task

- **Multi-resource** example
  - 2 resources: CPUs and mem
  - User 1 wants \(1CPU, 4GB\) per task
  - User 2 wants \(2CPU, 1GB\) per task

- What is a fair allocation?
Asset fairness: give weights to resources (e.g., 1 CPU = 1 GB) and equalize total value given to each user.
Asset fairness: give weights to resources (e.g., 1 CPU = 1 GB) and equalize total value given to each user.

Total resources: 28 CPU and 56 GB RAM (e.g., 1 CPU = 2 GB)

- User 1 has $x$ tasks and wants $\langle 1CPU, 2GB \rangle$ per task
- User 2 has $y$ tasks and wants $\langle 1CPU, 4GB \rangle$ per task
A Natural Policy (1/2)

- **Asset fairness**: give weights to resources (e.g., 1 CPU = 1 GB) and equalize total value given to each user.

- Total resources: 28 CPU and 56 GB RAM (e.g., 1 CPU = 2 GB)
  - User 1 has \(x\) tasks and wants \(\langle 1 \text{CPU}, 2 \text{GB} \rangle\) per task
  - User 2 has \(y\) tasks and wants \(\langle 1 \text{CPU}, 4 \text{GB} \rangle\) per task

- Asset fairness yields:

\[
\begin{align*}
\max(x, y) & \\
x + y & \leq 28 \\
2x + 4y & \leq 56 \\
4x & = 6y
\end{align*}
\]

User 1: \(x = 12\): \(\langle 43\% \text{CPU}, 43\% \text{GB} \rangle\) \((\sum = 86\% )\)
User 2: \(y = 8\): \(\langle 28\% \text{CPU}, 57\% \text{GB} \rangle\) \((\sum = 86\% )\)
- **Problem:** violates share grantee.

- User 1 gets less than 50% of both CPU and RAM.

- Better off in a separate cluster with half the resources.
Challenge

▶ Can we find a fair sharing policy that provides:
  • Share guarantee
  • Strategy-proofness

▶ Can we generalize max-min fairness to multiple resources?
Proposed Solution

Dominant Resource Fairness (DRF)
Dominant Resource Fairness (DRF) (1/2)

- **Dominant resource** of a user: the resource that user has the **biggest share of**.

  - Total resources: $\langle 8CPU, 5GB \rangle$
  - User 1 allocation: $\langle 2CPU, 1GB \rangle$
    
    \[
    \frac{2}{8} = 25\%CPU \text{ and } \frac{1}{5} = 20\%RAM
    \]
  - Dominant resource of User 1 is **CPU** ($25\% > 20\%$)
Dominant Resource Fairness (DRF) (1/2)

- **Dominant resource** of a user: the resource that user has the biggest share of.
  
  - Total resources: \( \langle 8 CPU, 5 GB \rangle \)
  - User 1 allocation: \( \langle 2 CPU, 1 GB \rangle \)
    \[ \frac{2}{8} = 25\% CPU \text{ and } \frac{1}{5} = 20\% RAM \]
  - Dominant resource of User 1 is CPU (25% > 20%)

- **Dominant share** of a user: the fraction of the dominant resource she is allocated.
  
  - User 1 dominant share is 25%.
Apply max-min fairness to dominant shares: give every user an equal share of her dominant resource.
Apply max-min fairness to dominant shares: give every user an equal share of her dominant resource.

Equalize the dominant share of the users.

- Total resources: $\langle 9 \text{CPU}, 18 \text{GB} \rangle$
- User 1 wants $\langle 1 \text{CPU}, 4 \text{GB} \rangle$; Dominant resource: RAM $\frac{1}{9} < \frac{4}{18}$
- User 2 wants $\langle 3 \text{CPU}, 1 \text{GB} \rangle$; Dominant resource: CPU $\frac{3}{9} > \frac{1}{18}$
Apply max-min fairness to dominant shares: give every user an equal share of her dominant resource.

Equalize the dominant share of the users.
- Total resources: \( \langle 9\text{CPU}, 18\text{GB} \rangle \)
- User 1 wants \( \langle 1\text{CPU}, 4\text{GB} \rangle \); Dominant resource: RAM \( \frac{1}{9} < \frac{4}{18} \)
- User 2 wants \( \langle 3\text{CPU}, 1\text{GB} \rangle \); Dominant resource: CPU \( \frac{3}{9} > \frac{1}{18} \)

\[
\begin{align*}
\text{max}(x, y) \\
x + 3y & \leq 9 \\
4x + y & \leq 18 \\
\frac{4x}{18} & = \frac{3y}{9} \\
\text{User 1: } x & = 3: \langle 33\%\text{CPU}, 66\%\text{GB} \rangle \\
\text{User 2: } y & = 2: \langle 66\%\text{CPU}, 16\%\text{GB} \rangle 
\end{align*}
\]
Whenever there are available resources and tasks to run:
Schedule a task to the user with the smallest dominant share.
YARN
YARN

Yet Another Resource Negotiator
YARN Architecture

- Resource Manager (RM)
- Application Master (AM)
- Node Manager (NM)
YARN Architecture - Resource Manager (1/2)

- **One per cluster**
  - Central: global view
  - Enable global properties
  - Fairness, capacity, locality

- **Job requests** are submitted to **RM**.
  - To start a job (application), RM finds a container to spawn AM.

- **Container**
  - Logical bundle of resources (CPU/memory).

- **No static resource partitioning.**
- Only handles an overall resource profile for each application.
  - Local optimization is up to the application.

- Preemption
  - Request resources back from an application.
  - Checkpoint snapshot instead of explicitly killing jobs / migrate computation to other containers.
The head of a job.

Runs as a container.

Request resources from RM.
- # of containers/resource per container/locality ...

Dynamically changing resource consumption, based on the containers it receives from the RM.
Requests are late-binding.
- The process spawned is not bound to the request, but to the lease.
- The conditions that caused the AM to issue the request may not remain true when it receives its resources.

Can run any user code, e.g., MapReduce, Spark, etc.

AM determines the semantics of the success or failure of the container.
YARN Architecture - Node Manager (1/2)

- The worker daemon.
- Registers with RM.
- One per node.
- Report resources to RM: memory, CPU, ...
- Containers are described by a Container Launch Context (CLC).
  - The command necessary to create the process
  - Environment variables
  - Security tokens
  - ...
▶ Configure the environment for task execution.

▶ Garbage collection.

▶ Auxiliary services.
  • A process may produce data that persist beyond the life of the container.
  • Output intermediate data between map and reduce tasks.
YARN Framework (1/2)

- **Submitting the application**: passing a CLC for the AM to the RM.

- When RM starts the AM, it should register with the RM.
  - Periodically advertise its liveness and requirements over the heartbeat protocol.
▶ Once the **RM** allocates a container, **AM** can construct a **CLC** to launch the container on the corresponding **NM**.
  • It **monitors** the status of the **running container** and stop it when the resource should be reclaimed.

▶ Once the **AM** is done with its work, it should unregister from the **RM** and **exit cleanly**.
Summary
Summary

- **Mesos**
  - Offered-based
  - Max-Min fairness: DRF

- **YARN**
  - Request-based
  - RM, AM, NM
Questions?

Acknowledgements

Some slides were derived from Ion Stoica and Ali Ghodsi slides (Berkeley University), and Wei-Chiu Chuang slides (Purdue University).