Distributed Systems 1

CUCS Course 4113
https://systems.cs.columbia.edu/ds1-class/

Instructor: Roxana Geambasu

Cluster Scheduling

Context

- We talked about cluster orchestration and Kubernetes.
- A key part of orchestration is *scheduling* (i.e., allocating compute resources to jobs -- nodes to pods in K8s).
- Now we'll talk about scheduling: a few common considerations, algorithms, and scheduler architectures in several real, open-source systems:
  - Apache YARN
  - Apache Mesos
  - Google Borg

Outline

Part 1: Cluster scheduling overview

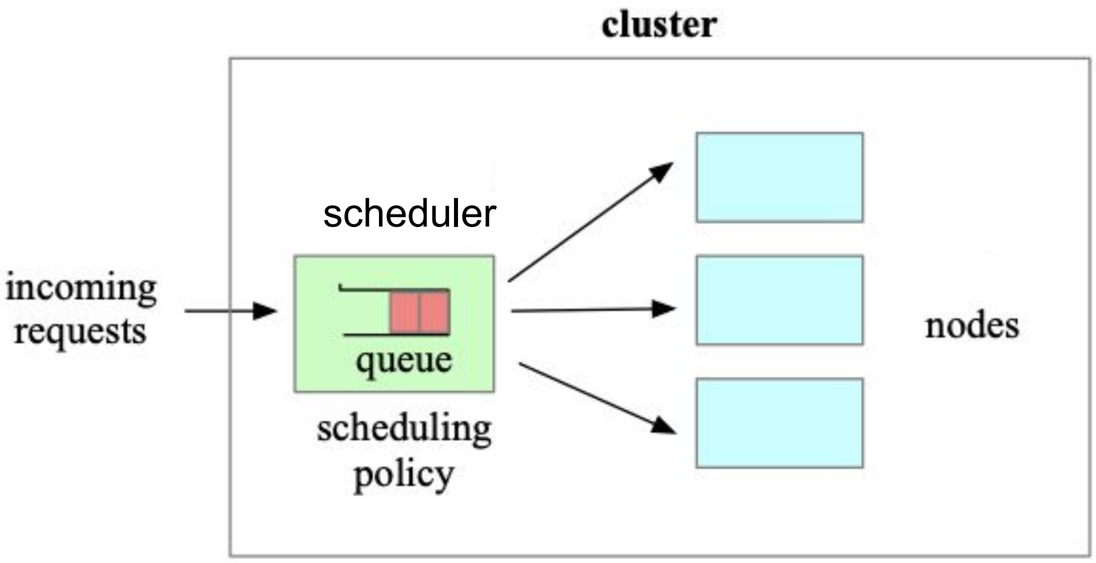
Part 2: Examples of real schedulers

Part 3: Scheduling algorithms

Part 1: Overview

Cluster Scheduling

- Scheduler (aka dispatcher) accepts incoming requests for _jobs_ and schedules them to run on nodes in the cluster.
- When to run and where to run each job are decisions made by the scheduler according to a scheduling policy.



Types of Workloads

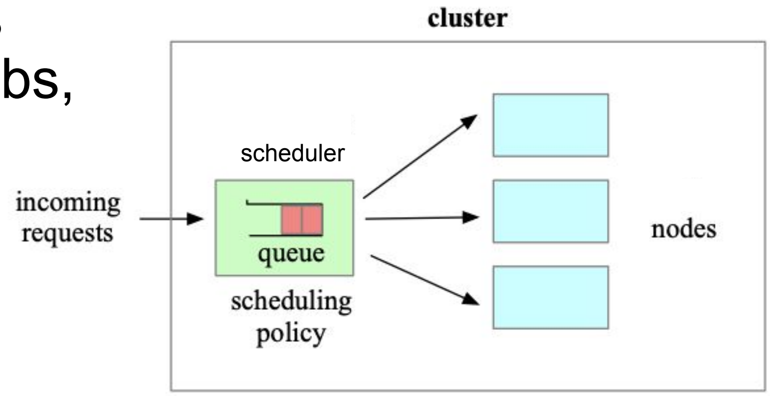
- **Interactive applications**
  - Submit short jobs whose response time must be short
  - E.g.: web service, where a “job” is a single HTTP request
  - Goal is to optimize for **response time**
- **Batch applications**
  - Job is a long running computation
  - Goal is to optimize for throughput
- Often **both types** of workloads share a cluster
  - Typically, prioritize interactive over batch

Scheduling in Interactive Applications

- Suppose you have a Web server deployment. How do you assign user requests to the servers in your deployment?
- Architecture:
  – N nodes: one node acts as **load balancer (LB)**, the others are replicas that constitute the **server pool**
  – HTTP requests arrive into queue at LB, which schedules each request onto a replica node
- How to decide <u>where</u> to run a given request? Scheduling policies: **least loaded**, **round robin**, **weighted round robin**
- How do decide <u>when</u> to service a given request? Typically **FIFO** (first in first out), but may prioritize users with established sessions
- Stateful services: LB at session level instead of per request

Scheduling Batch Jobs

- Batch jobs are non-interactive jobs
  - ML training, data processing jobs, indexing, simulations
- Scheduler architecture as before: users submit jobs to a queue, scheduler schedules them onto worker nodes



- Example: SLURM (Simple Linux Utility for Resource Management)
  - Runs on > 50% supercomputers
  - Nodes partitioned into groups; each group has job queue
  - Specify size, time limits, user groups for each queue
  - Many policies available: FIFO, priority, gang scheduling

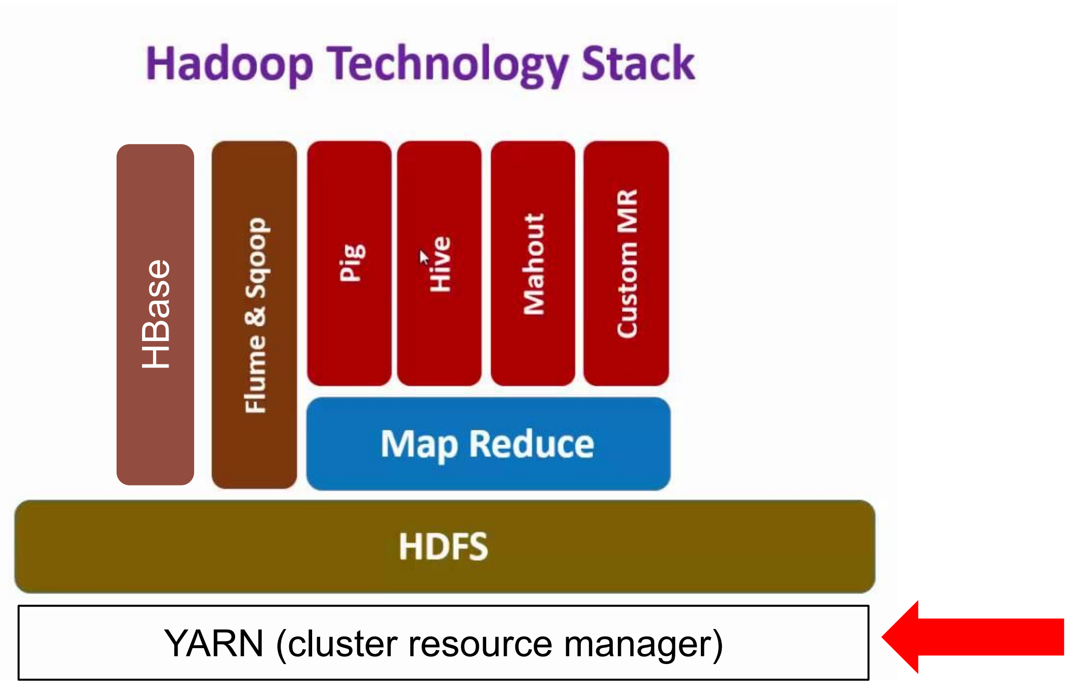
Part 2: Examples of Real Schedulers

Apache YARN

YARN

(“Yet Another Resource Negotiator”)

- Cluster manager typically used with **Apache Hadoop**
- Allocates resources to jobs to nodes in accordance a **scheduling policy**:
  - **FIFO**
  - **Capacity**
  - **Fair**



YARN Scheduling Policies

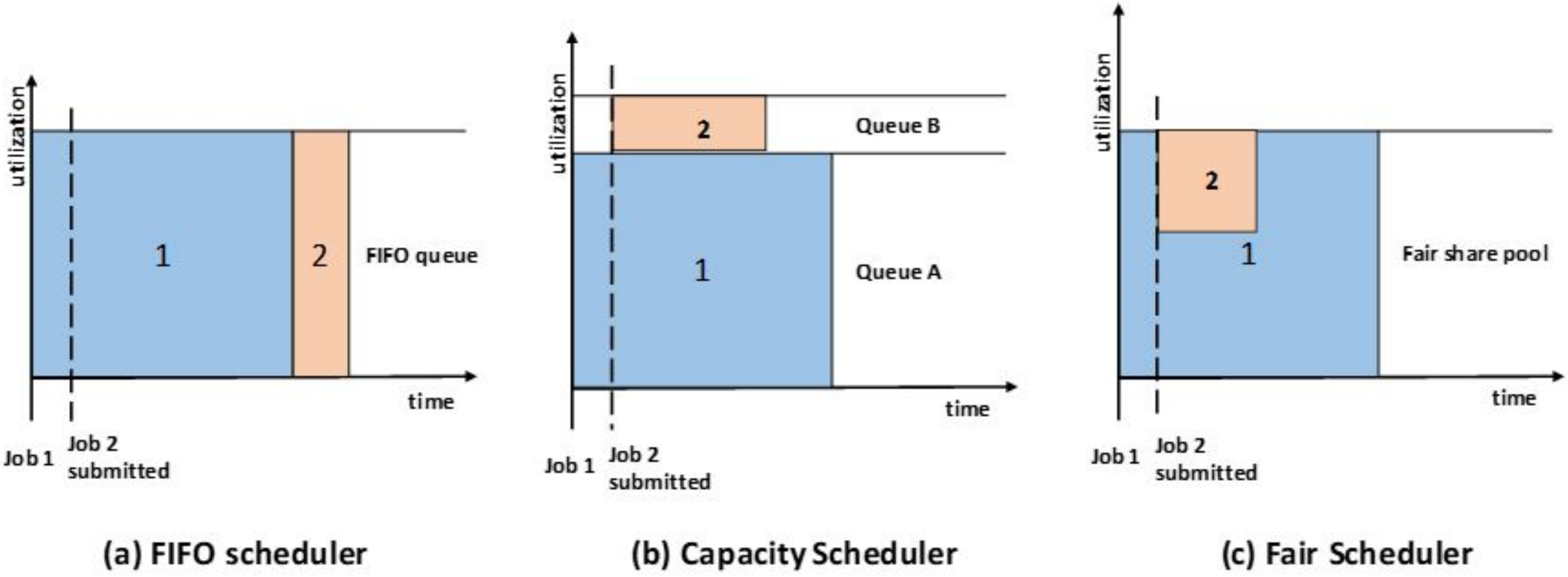


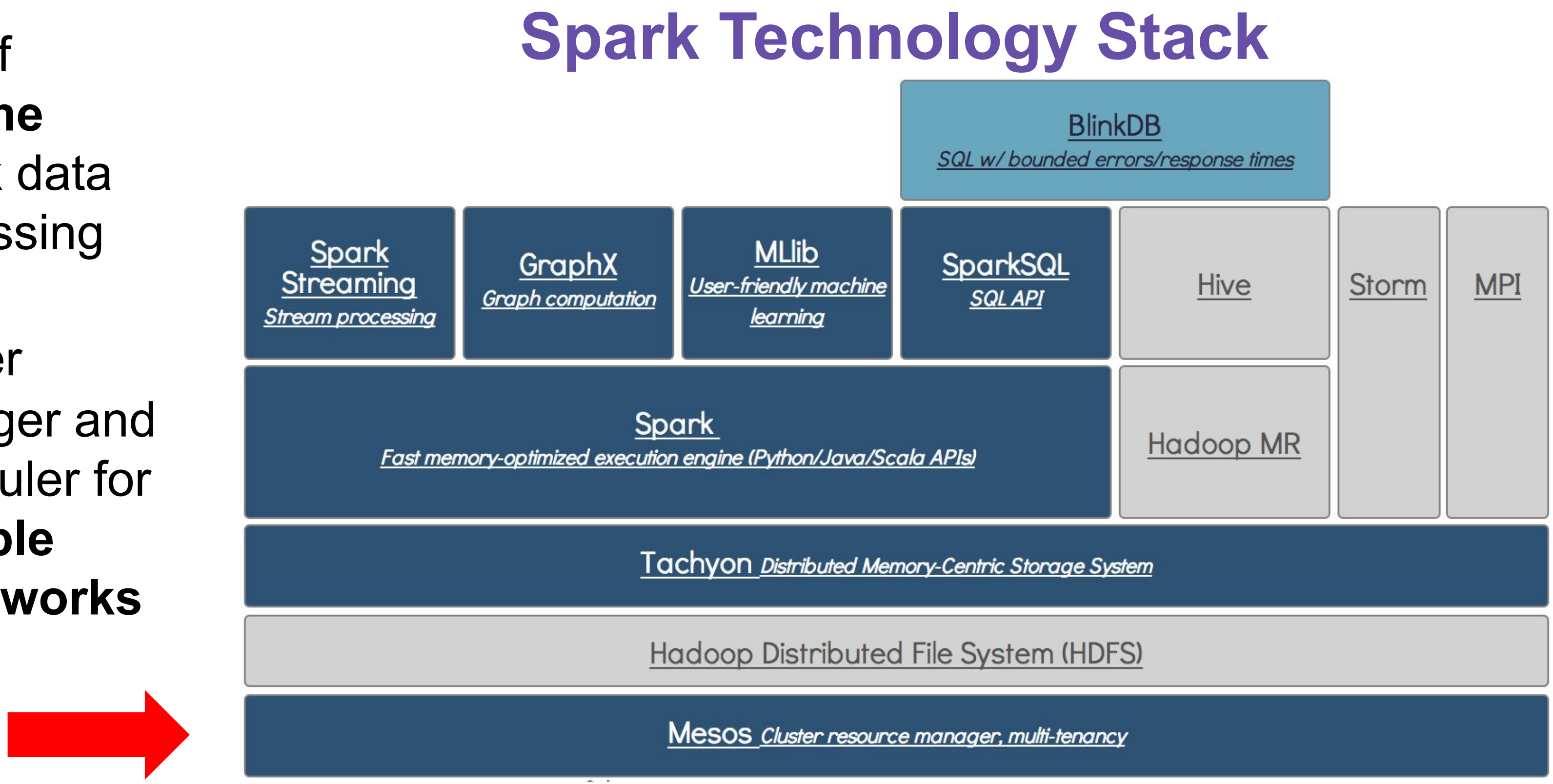
Figure 1: YARN Schedulers' cluster utilization vs. time

Part 2: Examples of Real Schedulers

Apache Mesos

Apache Mesos

- Part of **Apache Spark** data processing stack
- Cluster manager and scheduler for **multiple frameworks**



Mesos Overview

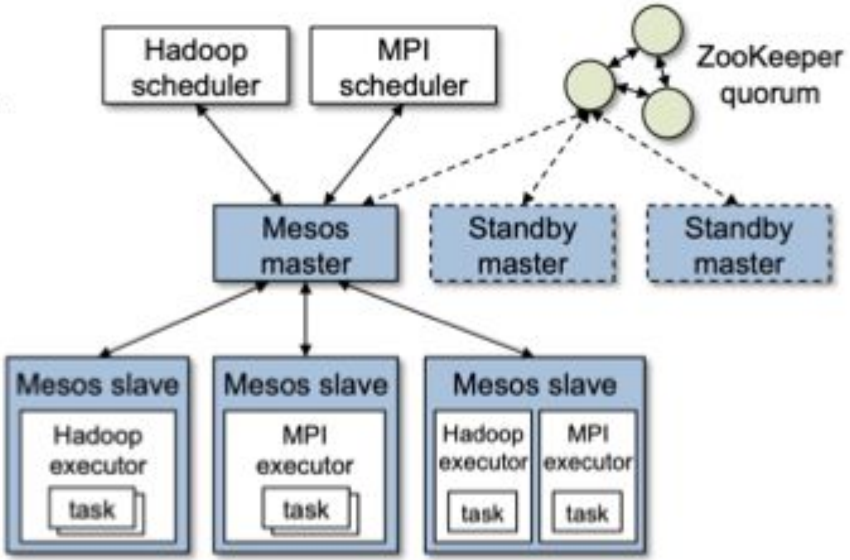
- Motivation: A cluster typically runs multiple frameworks -- Hadoop, Spark, MPI -- each with its scheduler. How should the cluster manage these frameworks?
  - One option: Statically partition cluster, each managed by a scheduler. Problem: fragments the cluster and may lead to under-utilization.

- **Mesos**: fine-grained server sharing between frameworks
  - Two-level approach: allocate resources to frameworks, framework allocates resources to jobs

- **Resource Offers**: bundle of resources offered to framework
  - Framework can accept or reject offer
  - Higher-level policy (e.g., fair share) governs allocation; resource offers used to offer resources
  - Framework-specific scheduling policy allocates to jobs
    - Framework can not ask for resources; only accept/reject resource offers

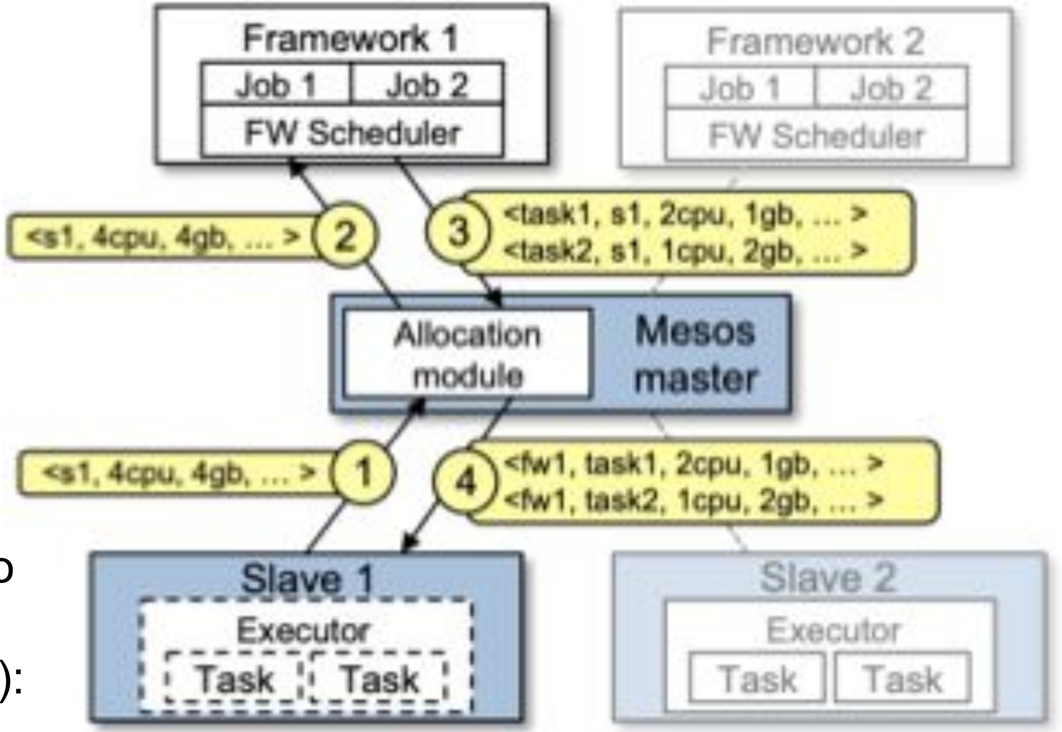
Mesos Architecture

Four components: coordinator, Mesos worker, framework scheduler, executor on server nodes



Example

- Step 1: worker node (4 core, 4GB) becomes idle, reports to coordinator
- Step 2: Coordinator invokes policy, decides to allocate to Framework 1. Sends resource offer
- Step 3: Framework accepts, scheduler assigns job 1 (2CPU, 1GB) and job 2 (1CPU, 2GB)
- Step 4: Coordinator sends job to executor on node
- Unused resources (1CPU, 1GB): new offer



Part 2: Examples of Real Schedulers

Google Borg

Google Borg

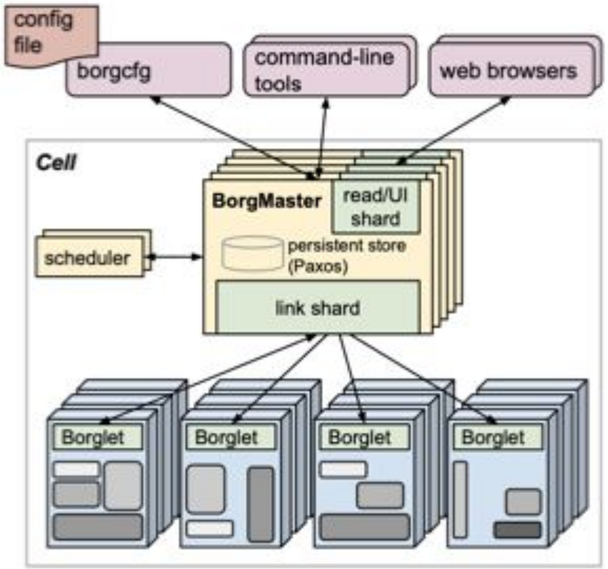
- Scheduling at very large scales: run hundreds of thousands of concurrent jobs onto tens of thousands of server
- Borg's ideas later influenced *kubernetes*
- Design Goals:
  - Hide details of resource management and failures from apps
  - Operate with high reliability (manages gmail, web search, ..)
  - Scale to very large clusters
- Designed to run two classes: interactive and batch
  – Long running interactive jobs (prod job) given priority
  – Batch jobs (non-prod jobs) given lower priority
  – % of interactive and batch jobs will vary over time

Borg Architecture

- Cell: group of machines in a cluster (~10K servers)
- Borg: matches jobs to cells
  – jobs specify resource needs
  – Borg finds a cell/machine to run a job
  – job needs can change (e.g., ask for more)

- Use resource reservations ("alloc")
  – alloc set: reservations across machines
  – Schedule job onto alloc set

- Preemption: higher priority job can preempt a lower priority job if there are insufficient resources
- Borg Master coordinator: replicated 5 times (paxos)
- Priority queue to schedule jobs: uses best-fit, worst-fit



Kubernetes Scheduler

- Some ideas come from Borg, but Kubernetes is more extensible and general
  - "You can customize the behavior of the `kube-scheduler` by writing a configuration file" (kube-scheduler documentation)

kube-scheduler documentation:

Extension points

Scheduling happens in a series of stages that are exposed through the following extension points:

1. `queueSort`: These plugins provide an ordering function that is used to sort pending Pods in the scheduling queue. Exactly one queue sort plugin may be enabled at a time.
2. `preFilter`: These plugins are used to pre-process or check information about a Pod or the cluster before filtering. They can mark a pod as unschedulable.
3. `filter`: These plugins are the equivalent of Predicates in a scheduling Policy and are used to filter out nodes that can not run the Pod. Filters are called in the configured order. A pod is marked as unschedulable if no nodes pass all the filters.
4. `postFilter`: These plugins are called in their configured order when no feasible nodes were found for the pod. If any `postFilter` plugin marks the Pod *schedulable*, the remaining plugins are not called.
5. `preScore`: This is an informational extension point that can be used for doing pre-scoring work.
6. `score`: These plugins provide a score to each node that has passed the filtering phase. The scheduler will then select the node with the highest weighted scores sum.
7. `reserve`: This is an informational extension point that notifies plugins when resources have been reserved for a given Pod. Plugins also implement an `Unreserve`

Part 3: Algorithms

Design Considerations

1. **Optimize for efficiency:**
   - Given fixed resources, run **as many jobs as possible** (or if jobs have different "utilities," get the highest global utility)
   - It's an instance of the **bin packing problem** (aka knapsack problem), which is NP-hard in general, but there are greedy approximations
2. **Ensure fairness:**
   - Given fixed resources, ensure that all jobs/users get, on average, an equal share of resources over time

Fairness and efficiency are often **at odds**, so choose one…

Utilization is often a secondary consideration alongside 1 or 2.

Part 3: Algorithms

Fairness-Oriented

Max-min Fairness

- Maximizes the minimum allocation received by a job (or a user).
- **Single-resource algorithm:**
  - Sort the job queue based on the share of the resource the jobs (or their users) have gotten so far.
  - Each time, you allocate a job/user at most its "fair share" of the resource (say CPU):

$$fs = R / N$$

where R is the capacity of the resource (e.g., number of CPU cores) and N is the number of jobs in the queue (or of users if we're doing max-min fairness at user level).

Properties

- **Sharing incentive:** Each user is better off sharing the cluster than exclusively using her own partition of the cluster.
  – Consider a cluster with identical nodes and N users. Then a user should not be able to allocate more jobs in a cluster partition consisting of 1/N of all resources.
- **Strategy proofness:** Users do not benefit by lying about their resource demands. This provides incentive compatibility.
- **Envy-freeness:** A user will not prefer the allocation of another user. This embodies fairness.
- **Pareto efficiency:** It is not possible to increase the allocation of a user without decreasing the allocation of at least another user. This maximizes **utilization** <u>subject to</u> satisfying the other properties.

(defs from DRF paper (see acks))

Multi-Resource Algorithm: DRF

- Max-min fairness described so far refers to only one resource.
- In reality, jobs request multiple resources, such as CPUs, memory, GPU, etc., and those demands are usually **heterogeneous**.
  - E.g., some jobs may request more CPU, others more memory; some request only CPUs and no GPUs, others request a combo; etc.
- **DRF (Dominant Resource Fairness)**: Algo to ensure max-min fairness across heterogeneous resource demands
  - E.g.: if user A runs CPU-heavy tasks and user B runs memory-heavy tasks, DRF attempts to equalize user A's share of CPUs with user B's share of memory.

DRF Algorithm

Algorithm 1 DRF pseudo-code

$R = \langle r_1, \cdots, r_m \rangle$ ▷ total resource capacities
$C = \langle c_1, \cdots, c_m \rangle$ ▷ consumed resources, initially 0
$s_i \ (i = 1..n)$ ▷ user i's dominant shares, initially 0
$U_i = \langle u_{i,1}, \cdots, u_{i,m} \rangle \ (i = 1..n)$ ▷ resources given to user i, initially 0

pick user i with lowest dominant share s_i
$D_i \leftarrow$ demand of user i's next task
if $C + D_i \leq R$ **then**
    $C = C + D_i$ ▷ update consumed vector
    $U_i = U_i + D_i$ ▷ update i's allocation vector
    $s_i = \max_{j=1}^{m}\{u_{i,j}/r_j\}$
else
    return ▷ the cluster is full
end if

dominant share = maximum share of any resource that has been granted to the user so far

(from DRF paper (see acks))

DRF Properties

- DRF enjoys the same game-theoretical properties of max-min fairness:
  - Sharing incentive
  - Strategy proofness
  - Envy-freeness
  - Pareto efficiency
- Thanks to these properties, it also ensures performance isolation among tasks/users.
- YARN includes it as one of its three options, and it's enabled by default in Cloudera's Hadoop stack.

Part 3: Algorithms

Efficiency-Oriented

Bin Packing

- Given fixed resources, run **as many jobs as possible** (or if jobs have different "utilities" or "weights," get the highest total weight)
- It's an instance of the **bin packing problem** (aka knapsack problem), which is NP-hard in general, expressed as the following ILP:

$$\max_{x_i \in \{0,1\}} \sum_{i=1} w_i x_i$$

$$\text{subject to} \quad \forall j \in [m] : \sum_{i=1}^{n} d_{ij} x_i \leq r_j$$

x_i = binary var (allocate or not job i)
w_i = weight/utility of job i
$d_{i,j}$ = demand of task i for resource j
r_j = capacity of resource j

Greedy Approximations

- Algo structure:
    - Sort jobs according to a ***task efficiency metric*** (**e_i**)
    - Allocate tasks in order, starting from the highest efficiency metric, until the algo cannot pack any more tasks
- Multiple definitions of **e_i**
    - For a single resource: $e_i = w_i/d_i$ (weight to demand ratio)
    - For multi-resource: $e_i = w_i/(\Sigma_{j=1..m} d_{i,j}/r_j)$
    - Others are possible, which underscore further the "scarcity" of a particular resource. No great agreement on best one, different schedulers make different choices

Acknowledgements

The lecture slides were inspired by the followings:

- Cluster scheduling lecture by Prof. Prashant Shenoy

  from UMass-Amherst
- YARN scheduler overview by Bilal Maqsood and documentation
- Kubernetes reference on scheduling
- DRF paper: Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, Ion Stoica. “Dominant Resource Fairness: Fair Allocation of Multiple Resource Types,” NSDI 2011.