TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Systems for Resource Management

Corso di Sistemi e Architetture per Big Data
A.A. 2022/23
Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack

High-level Interfaces

Data Processing

Data Storage

Resource Management

Support / Integration

Outline

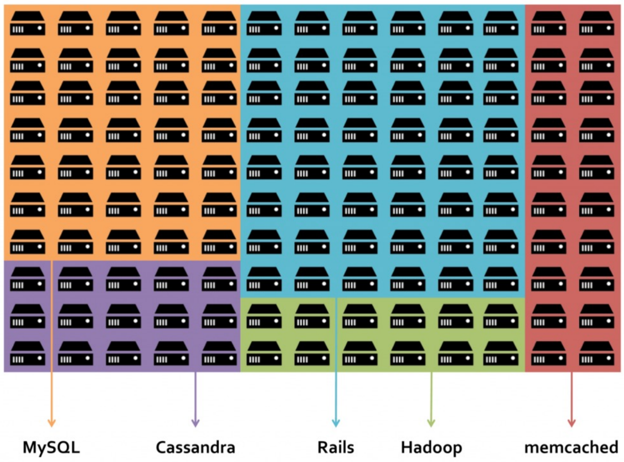
- Cluster management system
  – Apache Mesos
- Resource management policy
  – DRF

Why cluster resource management?

- Need to run multiple Big Data frameworks on same computing and storage infrastructure
- But running each framework on its dedicated cluster:
  ✗ Expensive
  ✗ Hard to share data
- Idea: share cluster resources among multiple Big Data frameworks

How to share: static partitioning

- How to share (virtual) cluster resources among multiple and heterogeneous Big Data frameworks?
- The simplest solution: static partitioning
- Efficient? No way

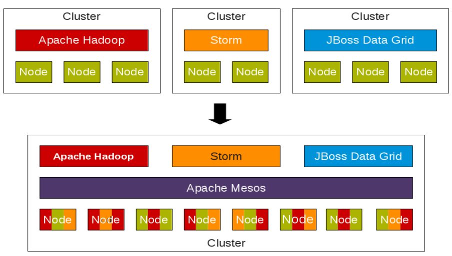


What we need

- *"The datacenter is the computer"* (D. Patterson)
  – Share resources to maximize their utilization
  – Share data among frameworks
  – Provide unified API to outside
  – Hide internal complexity of infrastructure from applications
- Solution: a cluster-scale resource manager that employs dynamic partitioning

Dynamic partitioning

☺



Apache Mesos

- Cluster manager that provides a common resource sharing layer over which diverse frameworks can run

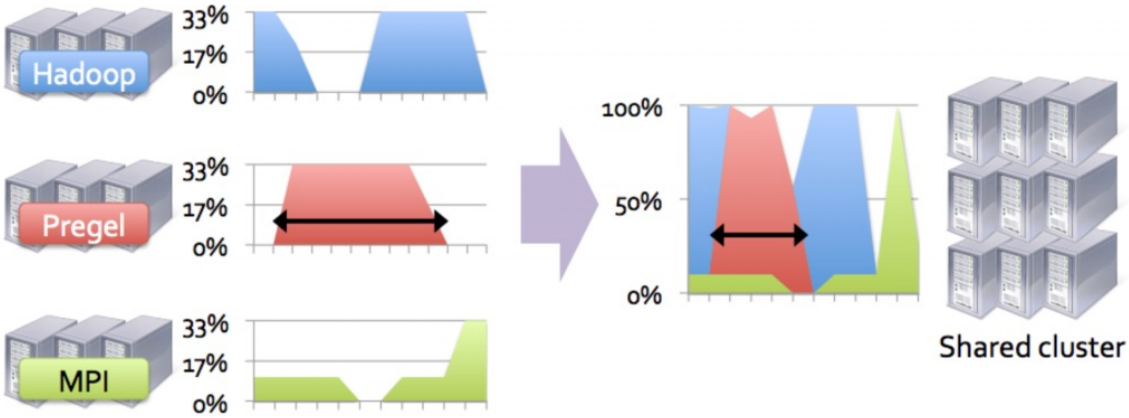
  “Program against your datacenter like it’s a single pool of resources” mesos.apache.org

  – Abstracts CPU, memory, storage, and other compute resources away from machines (physical or virtual), enabling fault-tolerant and elastic distributed systems to easily be built and run effectively

Apache Mesos

- Initially designed and developed at Berkeley Univ.
- Then Apache open-source project
- Used by many organizations (Airbnb, Twitter, Uber, Apple (Siri) among the others)
- Cluster as dynamically shared pool of resources

Static partitioning　　　　Dynamic partitioning

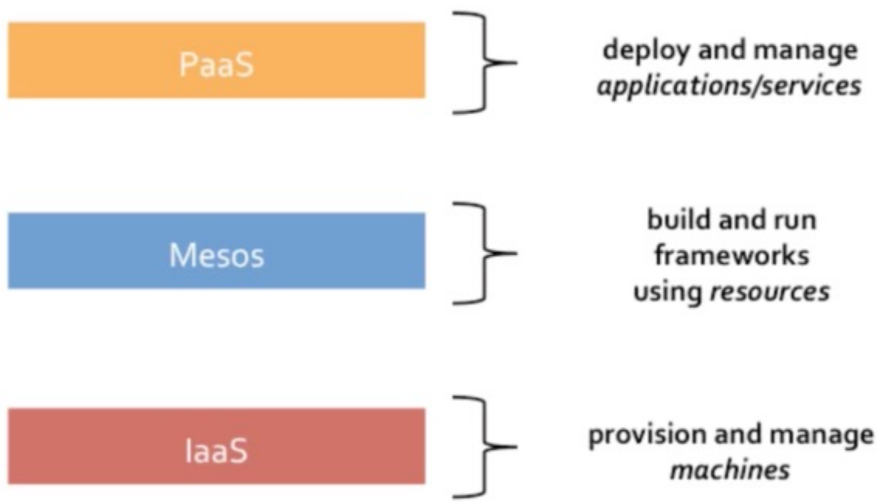


Mesos goals

- High utilization of resources
- Scalability to 10,000's of nodes
- High availability
- Support for many frameworks
  - But frameworks must be aware of running on Mesos
  - Which frameworks: mesos.apache.org/documentation/latest/frameworks/
    - Big Data processing: Hadoop, Flink, Spark, Storm
    - Data storage: Alluxio, Cassandra
    - Machine learning: TFMesos

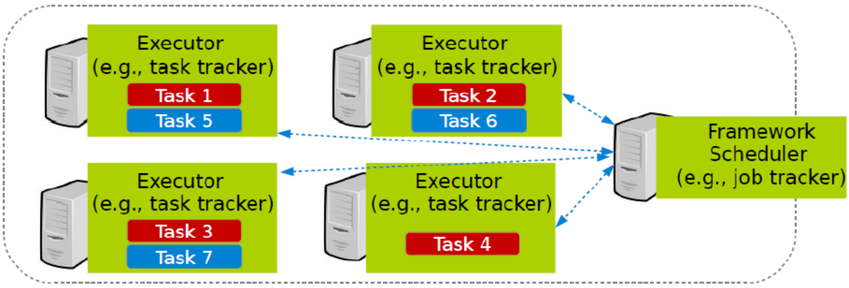
Mesos in the data center

- Where does Mesos fit as an abstraction layer in the datacenter?



Mesos computation model

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

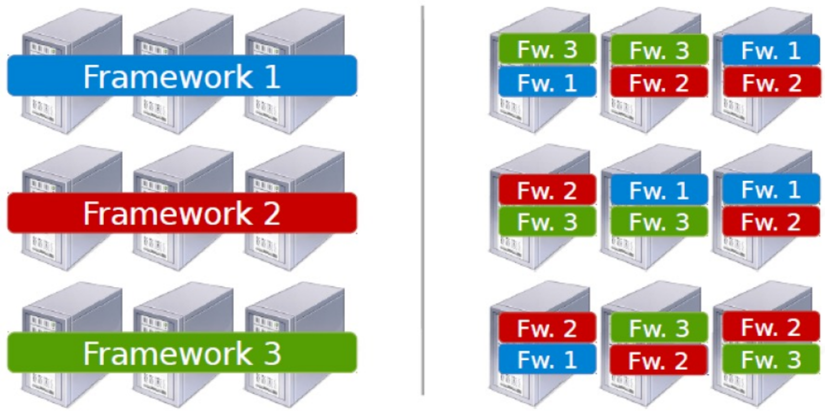


What Mesos does

- Enables fine-grained resource sharing (at the level of tasks within an application job) of resources (CPU, RAM, …) across frameworks
- Provides common functionalities:
  - Failure detection
  - Task distribution
  - Task starting
  - Task monitoring
  - Task killing
  - Task cleanup

Fine-grained sharing

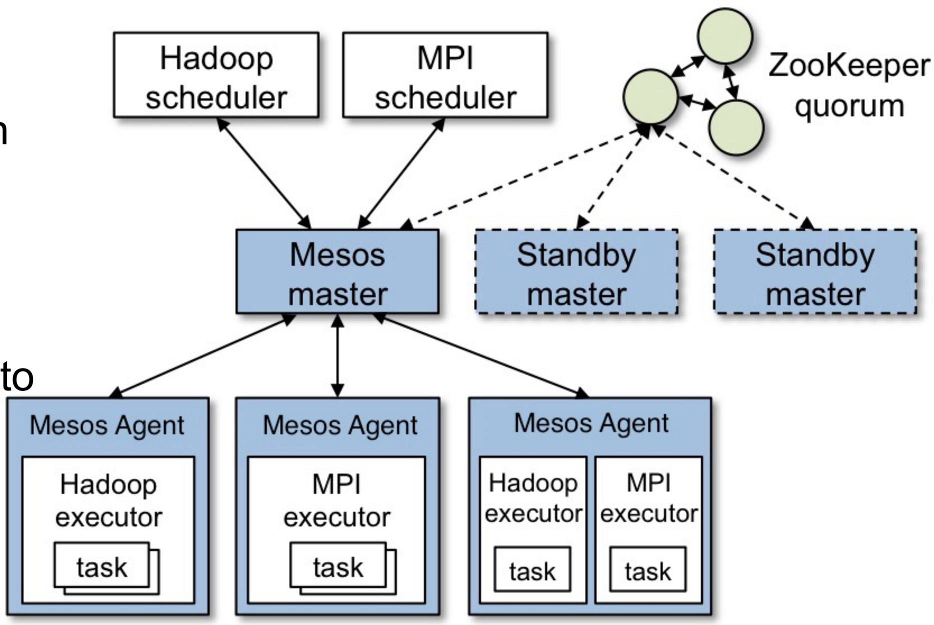
- Allocation at the level of tasks within a job
- Improves utilization, latency, and data locality



Coarse-grain sharing   Fine-grain sharing

Mesos: architecture

- Master-worker architecture
- Workers publish available resources to master
- Master sends resource offers to frameworks
- Master election and service discovery via ZooKeeper

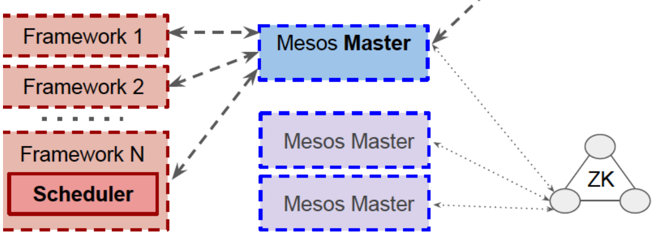


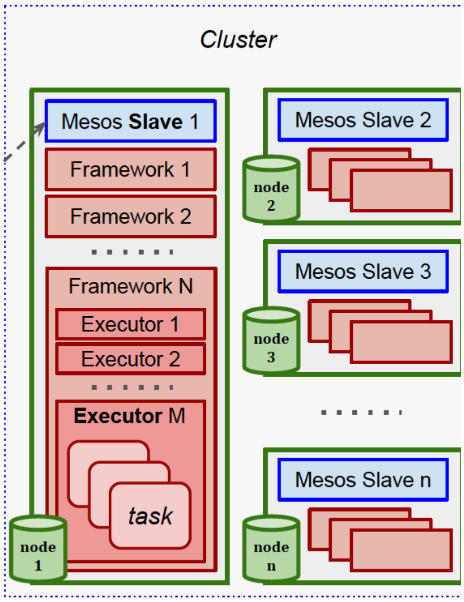
mesos.apache.org/documentation/latest/architecture/

Mesos: a platform for fine-grained resource sharing in the data center, NSDI'11

Mesos and framework components

- Mesos components
  - *Master*
  - *Workers* (or agents)
- Framework components
  - *Scheduler*: registers with Mesos master to accept resource offers
  - *Executors*: launched on agents to run framework's tasks

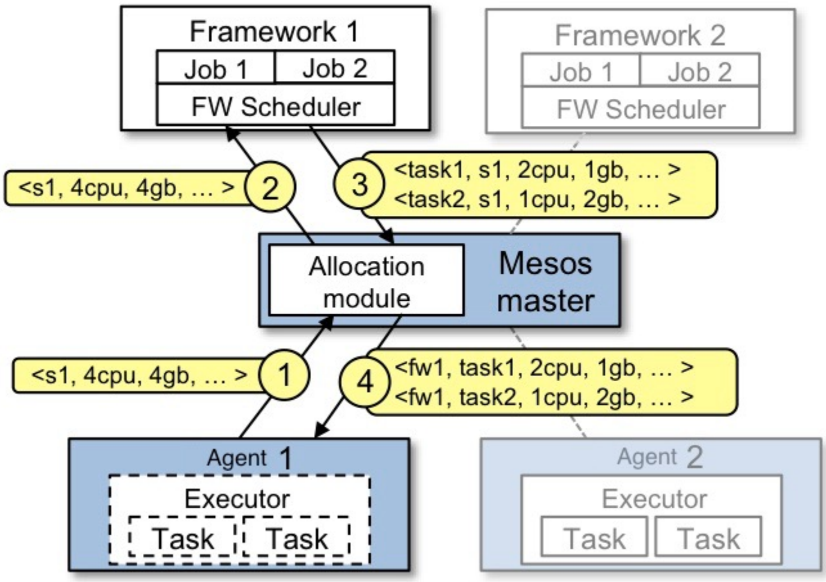




Scheduling in Mesos

- Scheduling based on resource offers
  - Mesos offers available resources to frameworks
    - Each resource offer contains a list of `<agent ID, resource1: amount1, resource2: amount2, ...>`
  - Each framework chooses which resources to use and which tasks to launch
- Two-level scheduler architecture
  - Mesos delegates the actual scheduling of tasks to frameworks
  - Why? To improve scalability
    - Master does not have to know the scheduling intricacies of every type of supported application

Mesos: resource offers



- We will see that resource allocation is based on Dominant Resource Fairness (DRF) algorithm

Mesos fault tolerance

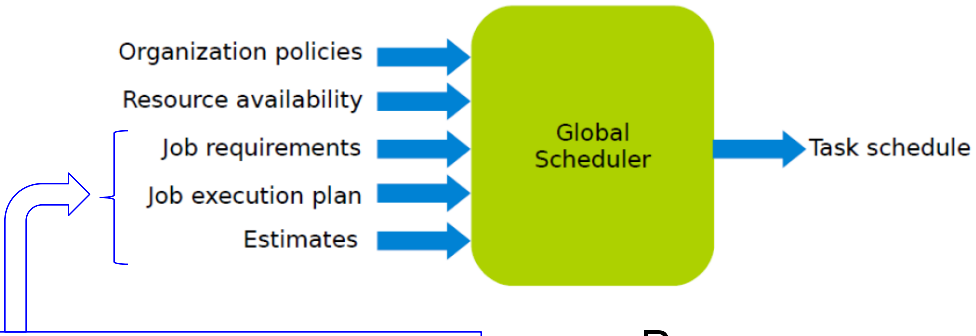
- Task failure
- Worker failure
- Host or network failure
- Master failure
- Framework scheduler failure

Cluster resource allocation

1. How to assign cluster resources to tasks?
   - Main design alternatives
     - Centralized scheduler
       - Global (monolithic) scheduler
       - Two-level scheduler
     - Decentralized scheduler
   - Let's focus on centralized scheduler

2. How to allocate resources of different types?

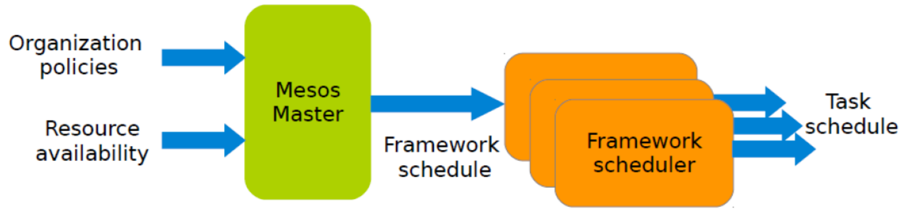
Global (monolithic) scheduler



- Job requirements
  - Response time, throughput, availability
- Job execution plan
  - Task DAG, inputs/outputs
- Estimates
  - Task duration, input sizes, transfer sizes

- Pros
  - Can achieve optimal schedule (global knowledge)
- Cons:
  - Complexity: hard to scale and ensure resilience
  - Hard to anticipate future frameworks requirements
  - Need to refactor existing frameworks

Two-level scheduler in Mesos

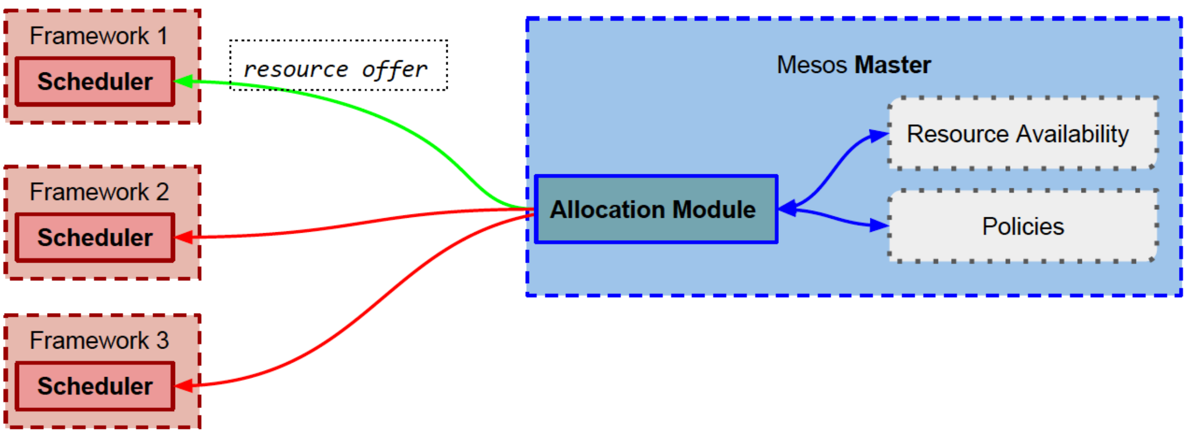


- Idea: push task placement to frameworks
- Resource offer
  - Vector of available resources on a node
  - E.g., node1: *<1CPU, 1GB>*, node2: *<4CPU, 16GB>*
- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

- Pros:
  - Simple: easier to scale and make resilient
  - Easy to port existing frameworks and support new ones
- Cons:
  - Two-level decision made by different entities: can be suboptimal

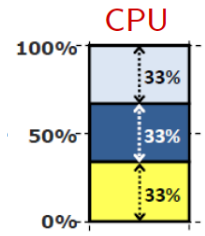
Mesos: resource allocation

- How to determine which frameworks to make resource offers?
- Dominant Resource Fairness (DRF) algorithm
  - Implemented by allocation module

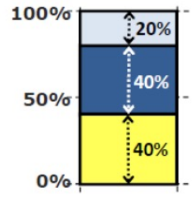


DRF: background on fair sharing

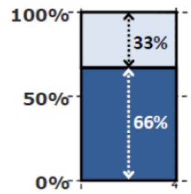
- Consider a single resource: fair sharing
  - *n* users want to share a resource, e.g., CPU
  - Solution: allocate each 1/*n* of the shared resource



- 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
  - Gives weights to users according to importance
  - E.g., user 1 gets weight 1, user 2 weight 2



Max-min fairness: example

- 1 resource type: CPU
- Total resources: 20 CPU
- User 1 has x tasks and wants <*1 CPU*> per task
- User 2 has y tasks and wants <*2 CPU*> per task

max(x, y) (maximize allocation)
subject to
$x + 2y \leq 20$ (CPU constraint)
$x = 2y$ (fairness)

Solution:
$x = 10$
$y = 5$

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 100
- Isolation policy
  - Users cannot affect others beyond their fair share

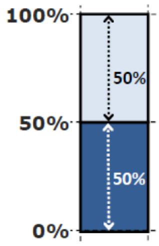
Why is fair sharing useful?

- Share guarantee
  - Each user can get at least 1/*n* of the resource
  - But will get less if its 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
- Many schedulers use max-min fairness
  - OS, networking, data centers (e.g., YARN)

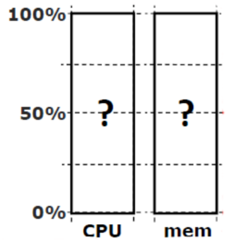
Max-min fairness drawback

- When is max-min fairness *not* enough?
- Need to schedule multiple, heterogeneous resources (CPU, memory, disk, I/O)

- 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 memory
  – User 1 wants <*1CPU, 4GB*> per task
  – User 2 wants <*3CPU, 1GB*> per task
- In the latter case which is a fair allocation?



What Mesos needs

- A fair sharing policy that provides:
  – Share guarantee
  – Strategy-proofness

- Challenge: can we generalize max-min fairness to multiple resources?

- Solution: Dominant Resource Fairness (DRF)

Dominant Resource Fairness: Fair Allocation of Multiple Resource Types, NSDI'11

DRF

- Dominant resource of a user: the resource that user has the biggest share of
  - Example:
    - Total resources: <8CPU, 5GB>
    - User 1 allocation: <2CPU, 1GB>
    - 2/8 = 25% CPU and 1/5 = 20% RAM
    - Dominant resource of user 1 is CPU (25% > 20%)
- Dominant share of a user: the fraction of the dominant resource allocated to the user
  - Example: User 1 dominant share is 25%
- DRF applies max-min fairness to dominant shares: give every user an equal share of its dominant resource
- Goal: equalize the dominant share of the users

DRF: example

- Goal: equalize the dominant share of the users
  - Total resources: <9CPU, 18GB>
  - User 1 wants <1CPU, 4GB>
  - Dominant resource for user 1: RAM (1/9 < 4/18)
  - User 2 wants <3CPU, 1GB>
  - Dominant resource for user 2: CPU (3/9 > 1/18)

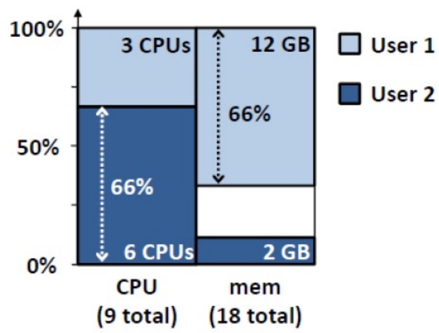
$\max(x, y)$

$x + 3y \le 9$

$4x + y \le 18$

$(4/18)x = (3/9)y$

- User 1: x = 3 <33%CPU, 66%GB>
- User 2: y = 2 <66%CPU, 16%GB>



Online DRF

- Whenever there are available resources and tasks to run:

  Choose the framework with the lowest dominant share among all frameworks

  – Online DRF tracks the total resources allocated to each user as well as the user's dominant share

  – At each step, DRF picks the user with the lowest dominant share among those with task ready to run

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

Schedule	User A		User B		CPU	RAM
	res. shares	dom. share	res. shares	dom. share	total alloc.	total alloc.
User B	$\langle 0, 0 \rangle$	**0**	$\langle 3/9, 1/18 \rangle$	1/3	3/9	1/18
User A	$\langle 1/9, 4/18 \rangle$	**2/9**	$\langle 3/9, 1/18 \rangle$	1/3	4/9	5/18
User A	$\langle 2/9, 8/18 \rangle$	4/9	$\langle 3/9, 1/18 \rangle$	**1/3**	5/9	9/18
User B	$\langle 2/9, 8/18 \rangle$	**4/9**	$\langle 6/9, 2/18 \rangle$	2/3	8/9	10/18
User A	$\langle 3/9, 12/18 \rangle$	**2/3**	$\langle 6/9, 2/18 \rangle$	**2/3**	1	14/18

DRF: efficiency-fairness trade-off

- DRF causes under-utilized resources
- DRF schedules at the level of tasks (leads to sub-optimal job completion time)
- Fairness is fundamentally at odds with overall efficiency (how to trade-off?)