TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

Systems for Resource Management

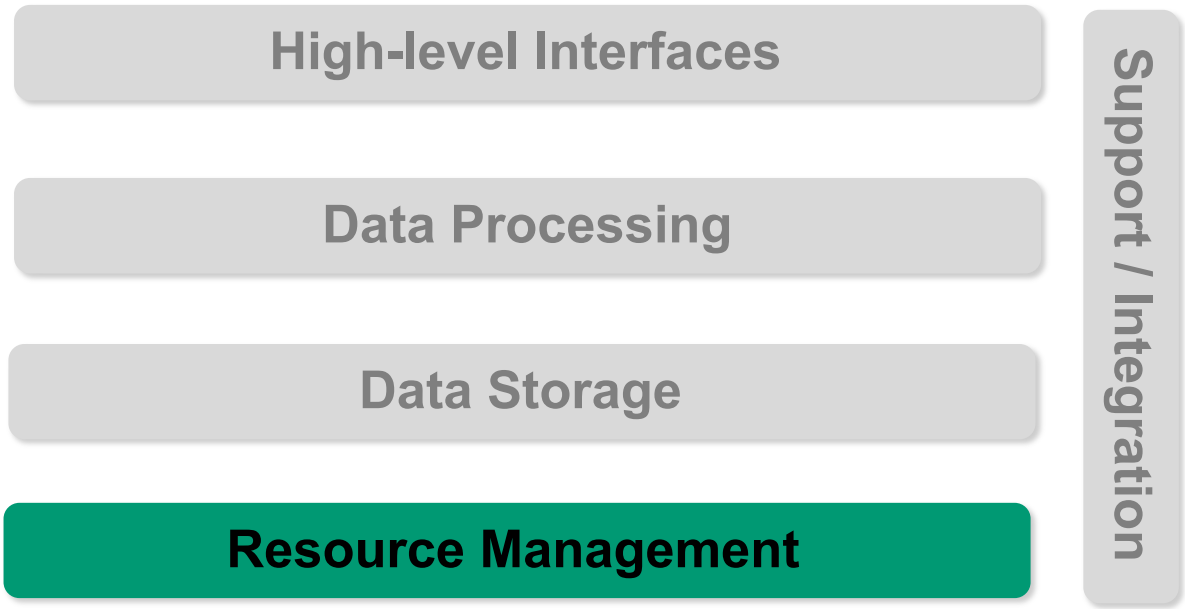
Corso di Sistemi e Architetture per Big Data

A.A. 2021/22

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack



Outline

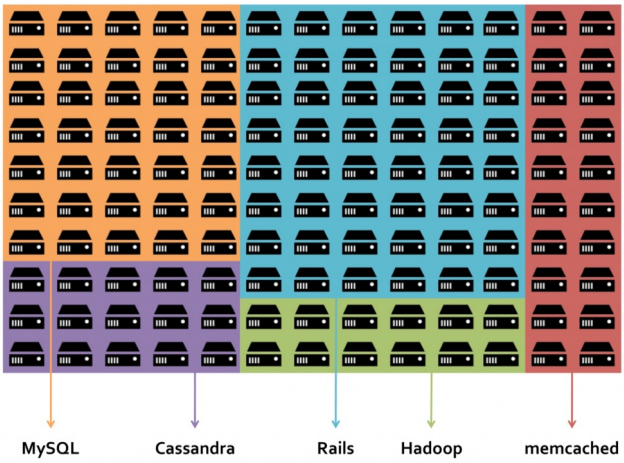
- Cluster management system
  - Mesos
- Resource management policy
  - DRF

Motivations

- Need to run multiple Big Data frameworks on same infrastructure
- 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 non homogeneous Big Data frameworks executed in virtual machines/containers?
- Baseline 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 a unified API to the outside
  - Hide the internal complexity of the infrastructure from applications
- Solution: a cluster-scale resource manager that employs dynamic partitioning

Apache Mesos

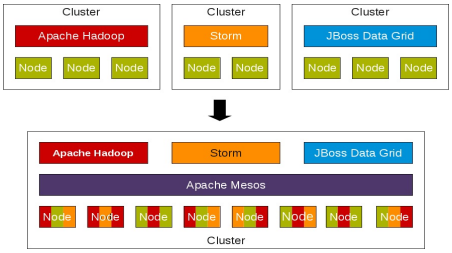
- Cluster manager that provides a common resource sharing layer over which diverse frameworks can run

  "Program against your datacenter like it's a single pool of resources" https://mesos.apache.org

  – Abstracts the entire datacenter into a single pool of computing resources, simplifying running distributed systems at scale

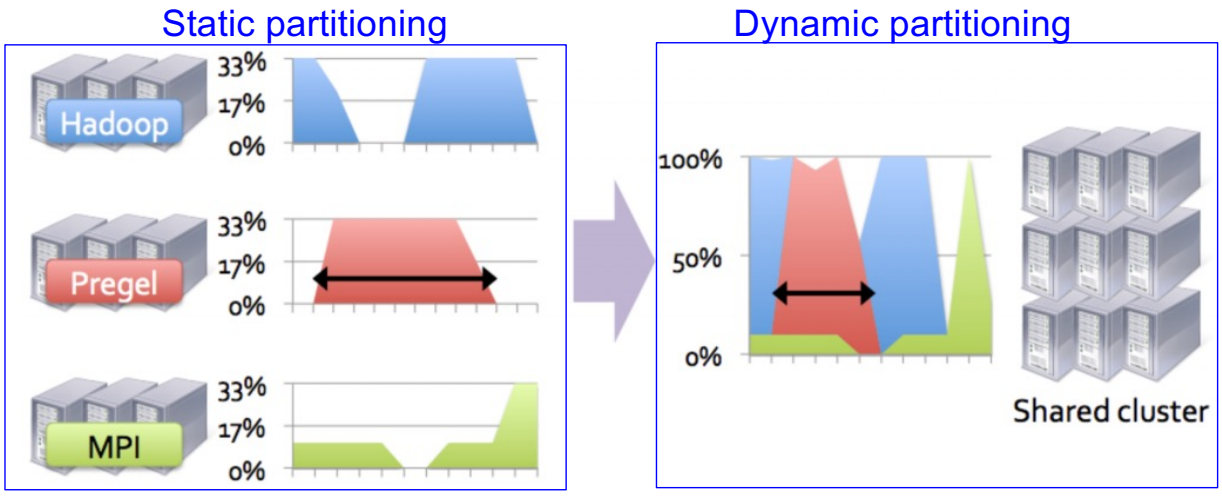
  – Distributed system to build and run fault-tolerant and elastic distributed systems on top of it

Dynamic partitioning



Apache Mesos

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

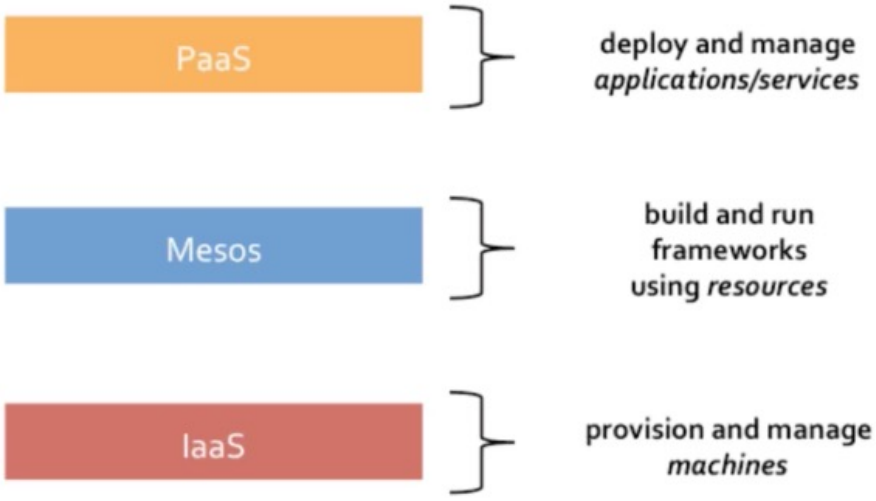


Mesos goals

- High utilization of resources
- Support for diverse frameworks (current and future)
- Scalability to 10,000's of nodes
- Reliability in face of failures

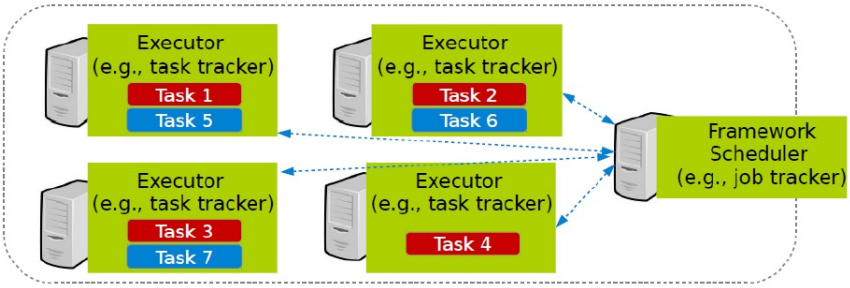
Mesos in the data center

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



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

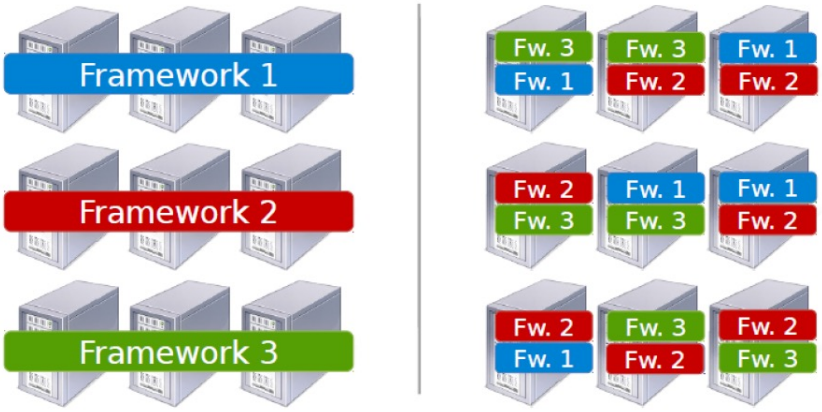


What Mesos does

- Enables fine-grained resource sharing (at the level of tasks within a 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

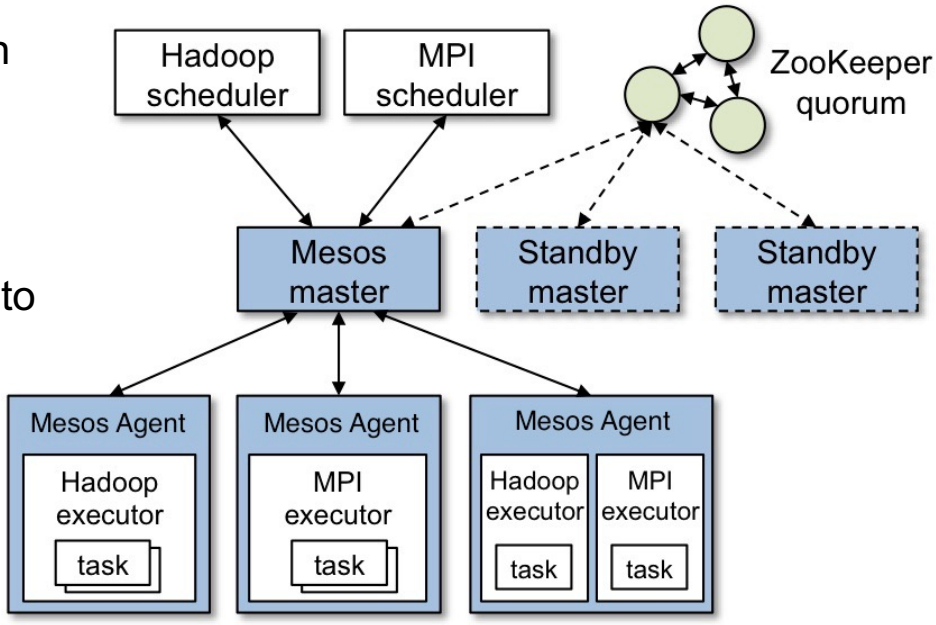
Frameworks on Mesos

- Frameworks must be aware of running on Mesos
  - DevOps tooling: Vamp
    - Deployment and workflow tool for container orchestration
  - Long running services: Aurora (service scheduler), …
  - Big Data processing: Hadoop, Flink, Spark, Storm, …
  - Batch scheduling: Chronos, …
  - Data storage: Alluxio, Cassandra, ElasticSearch, …
  - Machine learning: TFMesos
    - Framework to help running distributed Tensorflow ML tasks on Apache Mesos with GPU support

Full list at mesos.apache.org/documentation/latest/frameworks/

Mesos: architecture

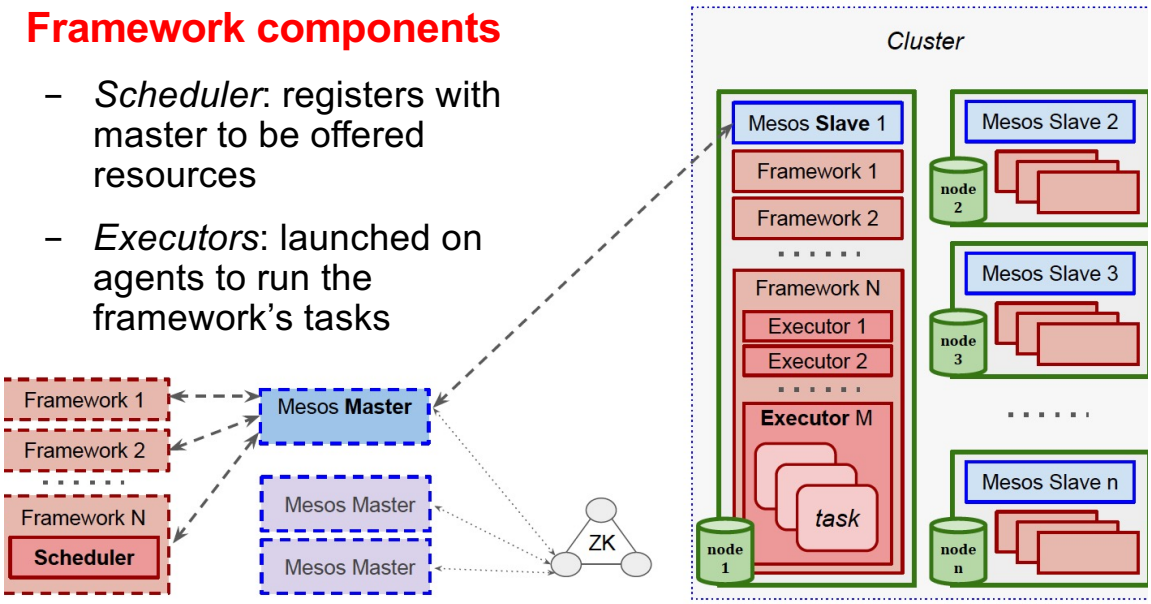
- Master-worker architecture
- Workers publish available resources to master
- Master sends resource offers to frameworks
- Master election and service discovery via ZooKeeper



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

Mesos and framework components

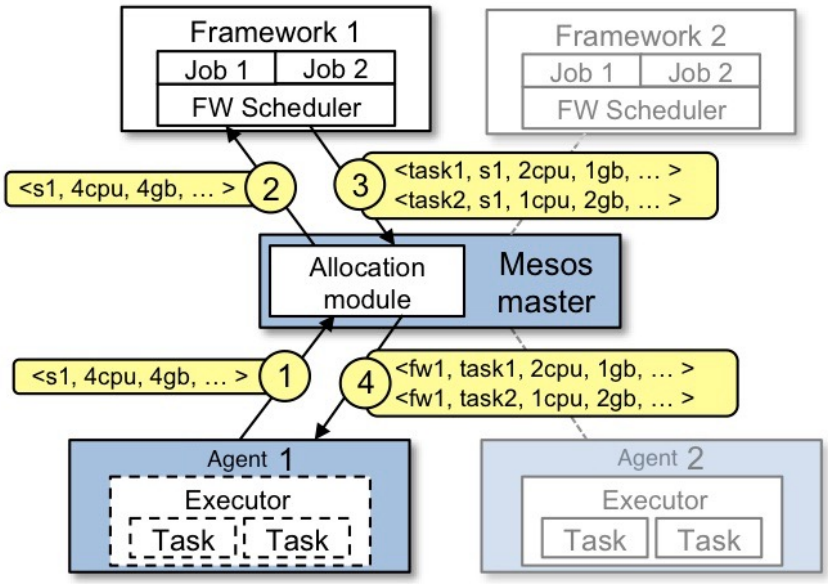
- **Mesos components**
  - *Master*
  - *Workers* or agents
- **Framework components**
  - *Scheduler*: registers with master to be offered resources
  - *Executors*: launched on agents to run the framework's tasks



Scheduling in Mesos

- Scheduling mechanism 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

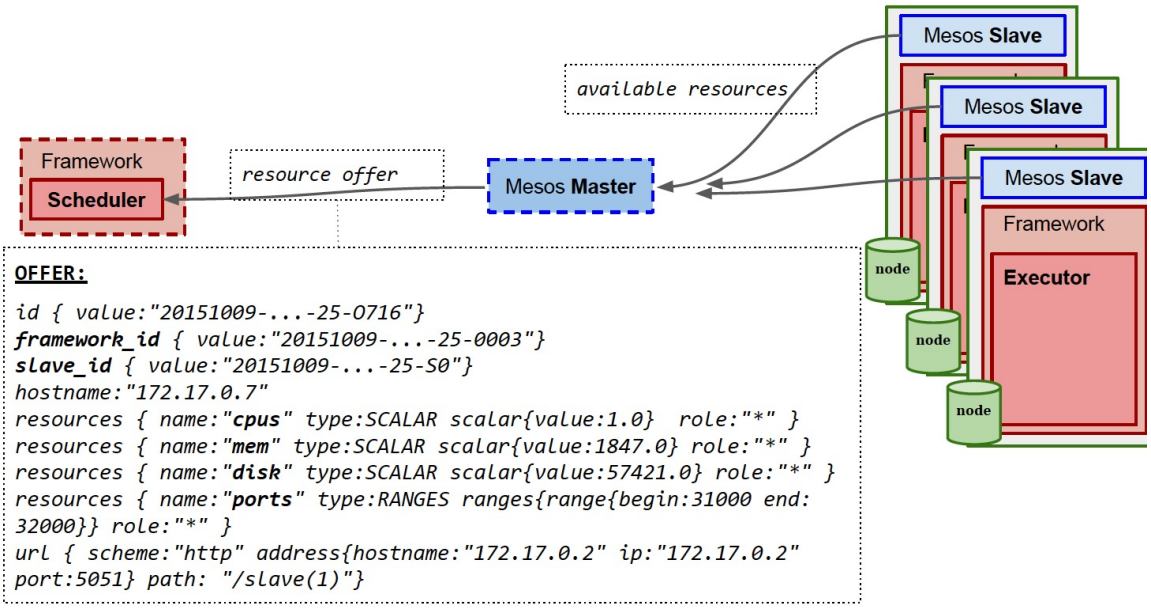
Mesos: resource offers



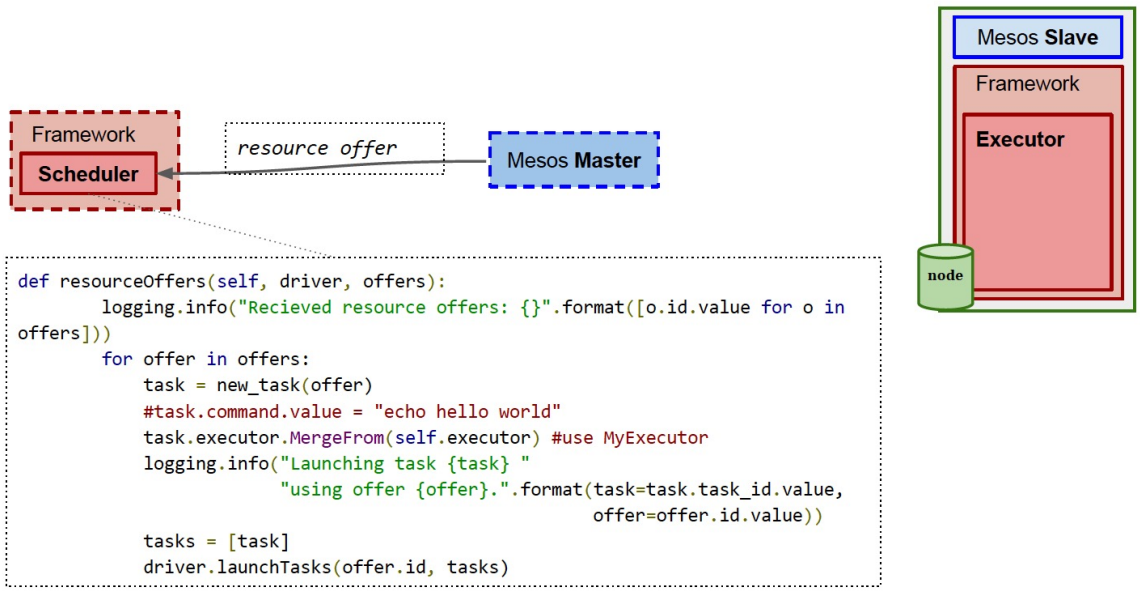
- Resource allocation is based on Dominant Resource Fairness (DRF) algorithm

Mesos: resource offers in details

- Workers continuously send status updates about resources to master

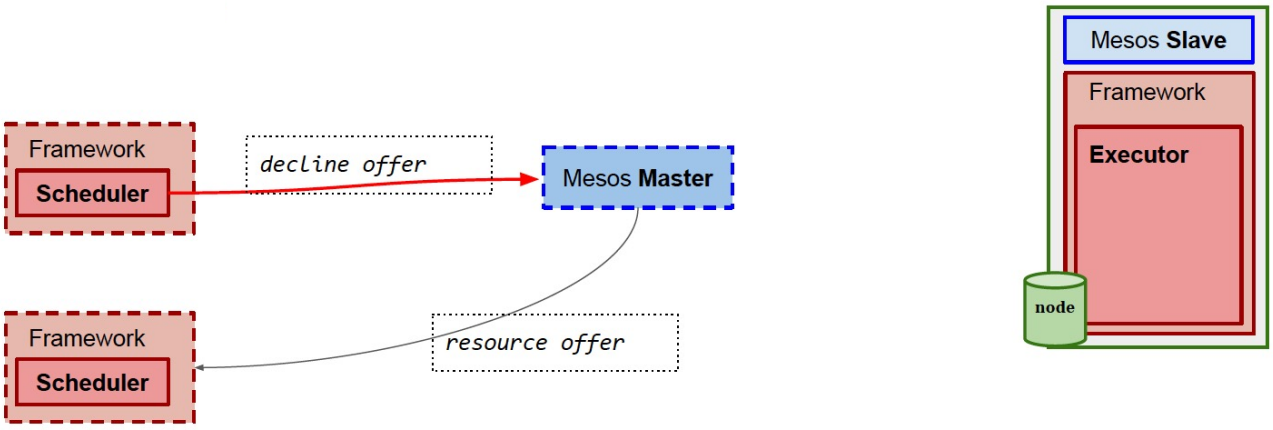


Mesos: resource offers in details (2)



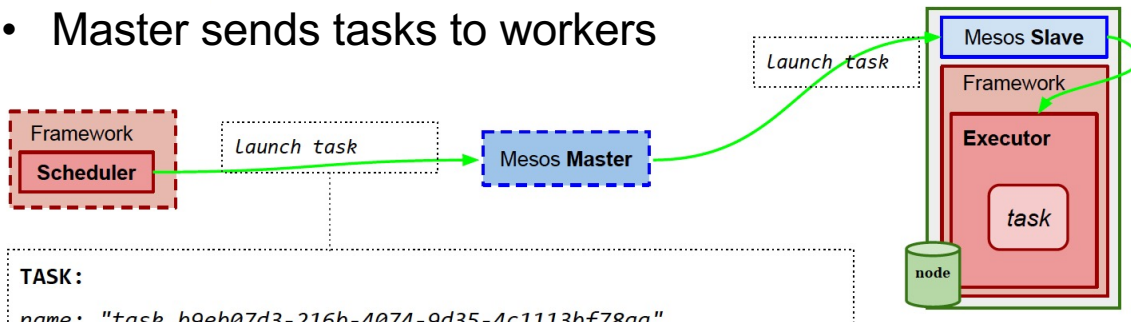
Mesos: resource offers in details (3)

- Framework scheduler can **reject** offers



Mesos: resource offers in details (4)

- Framework scheduler selects resources and provides tasks
- Master sends tasks to workers

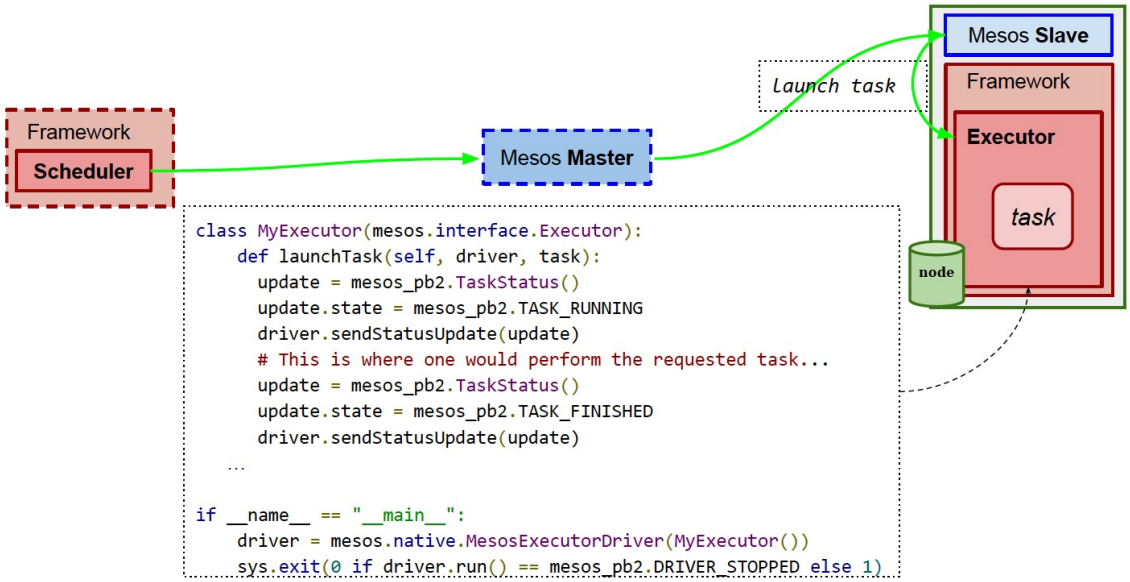


TASK:

```
name: "task b9eb07d3-216b-4074-9d35-4c1113bf78aa"
task_id {value:"b9eb07d3-216b-4074-9d35-4c1113bf78aa"}
slave_id {value:"20151009-082055-16781740-5050-25-S2"}
resources {name:"cpus" type:SCALAR scalar{value:1}}
resources {name:"mem" type:SCALAR scalar{value:128}}
executor {
     executor_id{value:"my_exec_ID_8945763725"}
     command {value:"python /SF/python/test/test_executor.py"}
}
```

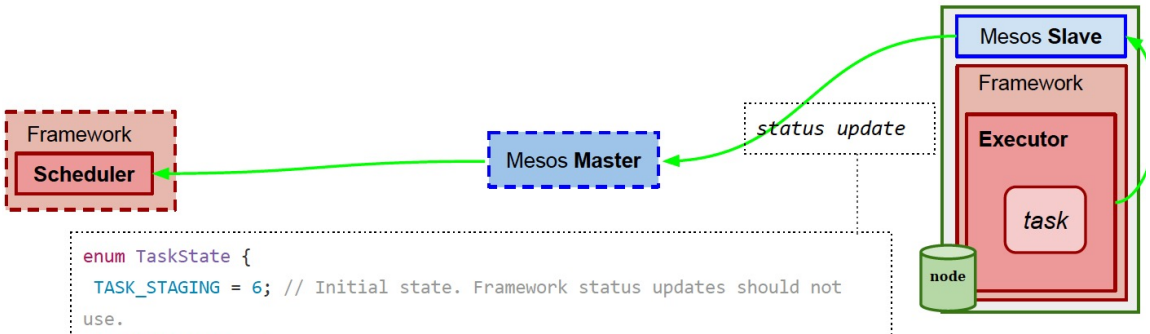
Mesos: resource offers in details (5)

- Framework executors launch tasks



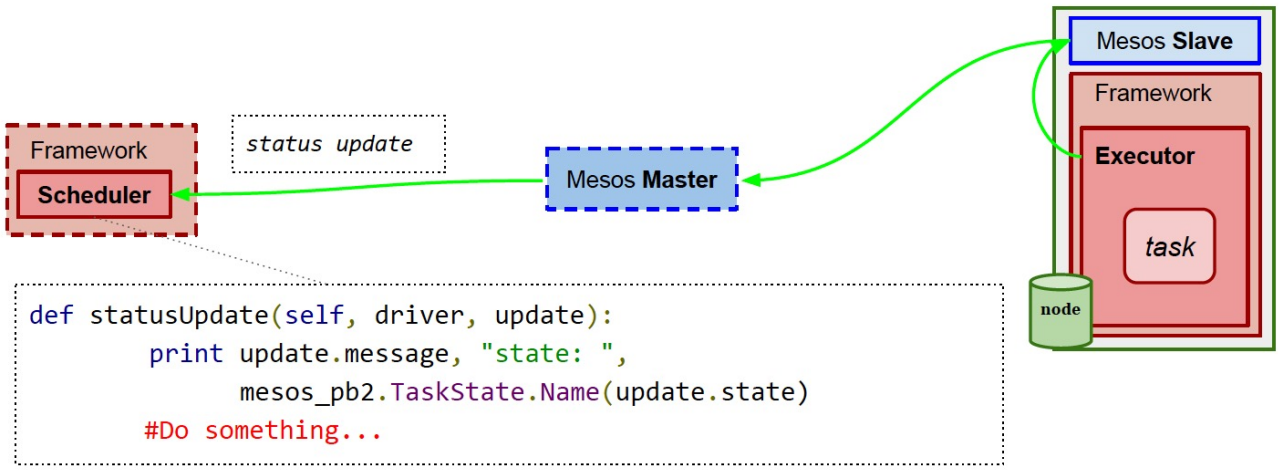

```
class MyExecutor(mesos.interface.Executor):
    def launchTask(self, driver, task):
      update = mesos_pb2.TaskStatus()
      update.state = mesos_pb2.TASK_RUNNING
      driver.sendStatusUpdate(update)
      # This is where one would perform the requested task...
      update = mesos_pb2.TaskStatus()
      update.state = mesos_pb2.TASK_FINISHED
      driver.sendStatusUpdate(update)
   ...

if __name__ == "__main__":
    driver = mesos.native.MesosExecutorDriver(MyExecutor())
    sys.exit(0 if driver.run() == mesos_pb2.DRIVER_STOPPED else 1)
```


Mesos: resource offers in details (6)




```
enum TaskState {
 TASK_STAGING = 6; // Initial state. Framework status updates should not
use.
 TASK_STARTING = 0;
 TASK_RUNNING = 1;
 TASK_FINISHED = 2; // TERMINAL. The task finished successfully.
 TASK_FAILED = 3; // TERMINAL. The task failed to finish successfully.
 TASK_KILLED = 4; // TERMINAL. The task was killed by the executor.
 TASK_LOST = 5; // TERMINAL. The task failed but can be rescheduled.
 TASK_ERROR = 7; // TERMINAL. The task description contains an error.
}
```

Mesos: resource offers in details (7)




```
def statusUpdate(self, driver, update):
        print update.message, "state: ",
              mesos_pb2.TaskState.Name(update.state)
        #Do something...
```

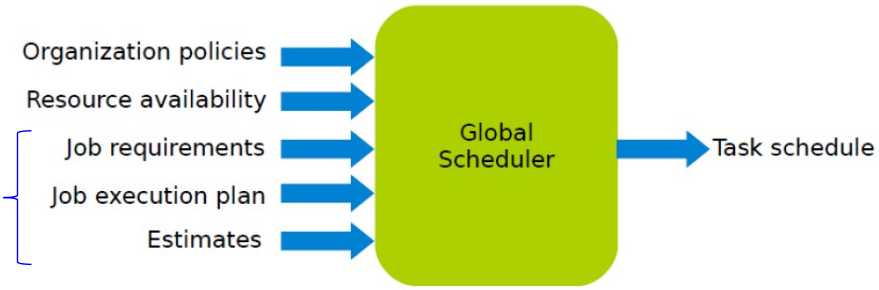

Mesos fault tolerance

- 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
     - Fully decentralized scheduler
   - Let's focus on centralized scheduler
2. How to allocate resources of different types?

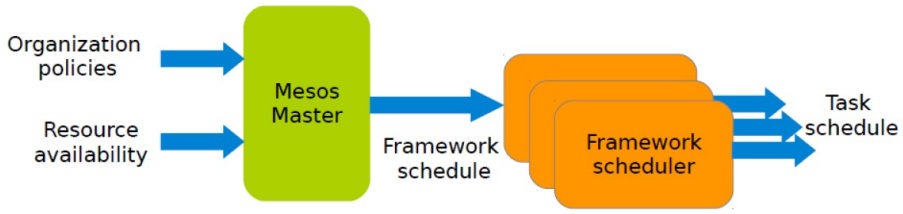
Global (monolithic) scheduler



- 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

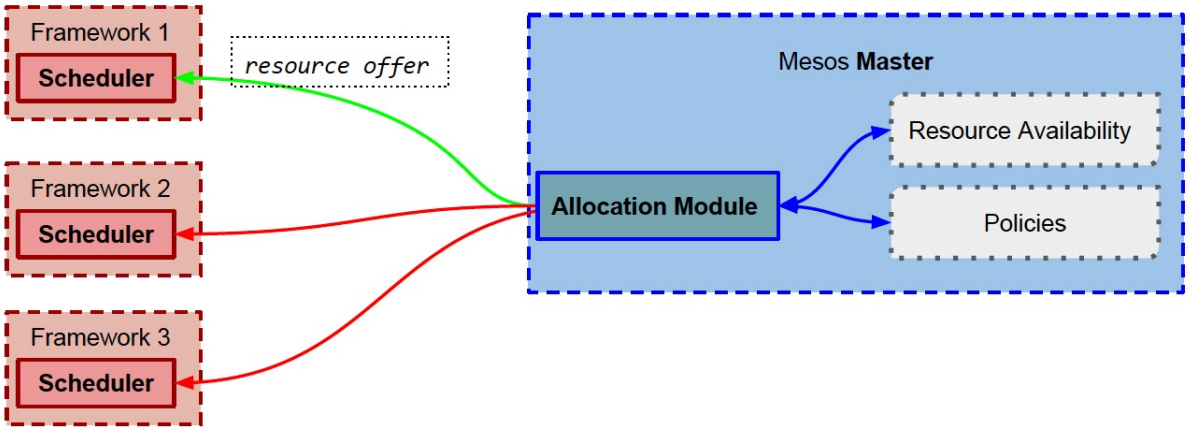


- Idea: push task placement to frameworks
- Resource offer
  - Vector of available resources on a node
  - E.g., node1: *<1CPU, 1GB>*, node2: *<4CPU, 16GB>*
- Master sends resource offers to frameworks
- Frameworks select which offers to accept and which tasks to run

- 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

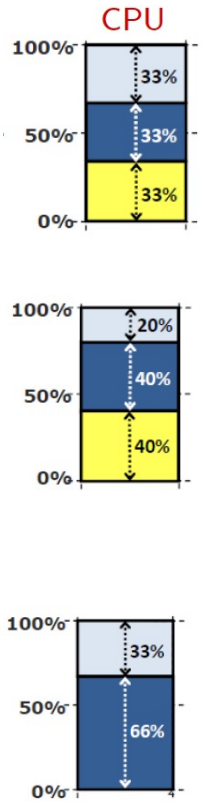
Mesos: resource allocation

- How to determine which frameworks to make resource offers?
- **Dominant Resource Fairness (DRF)** algorithm
  - Implemented in the 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 *<1CPU>* per task
- User 2 has y tasks and wants *<2CPU>* per task

max(x, y) (maximize allocation)
subject to
x + 2y ≤ 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? (2)

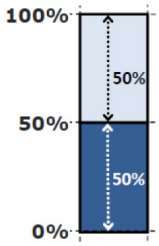
- Share guarantee
  - Each user can get at least 1/*n* of the resource
  - But will get less if its demand is less
- Strategy-proof
  - Users are not better off by asking for more than they need
  - Users have no reason to lie

- Max-min fairness is the only reasonable mechanism with these two properties
- Many schedulers use max-min fairness
  - OS, networking, datacenters (e.g., YARN)

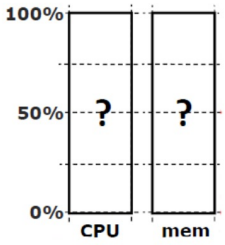
Max-min fairness drawback

- 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 what is a fair allocation?



A first (wrong) solution

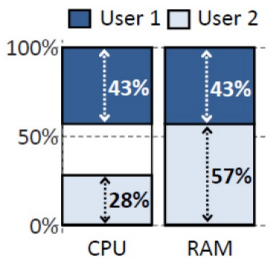
- Asset fairness: gives weights to resources (e.g., 1 CPU = 1 GB) and equalizes total allocation (i.e., sum) to each user
- Total resources: 28 CPUs and 56GB RAM (e.g., 1 CPU = 2 GB)
  - User 1 has x tasks and wants <*1CPU, 2GB*> per task
  - User 2 has y tasks and wants <*1CPU, 4GB*> per task
- Asset fairness yields:

  max(x, y) (maximize allocation)

  x + y ≤ 28

  2x + 4y ≤ 56

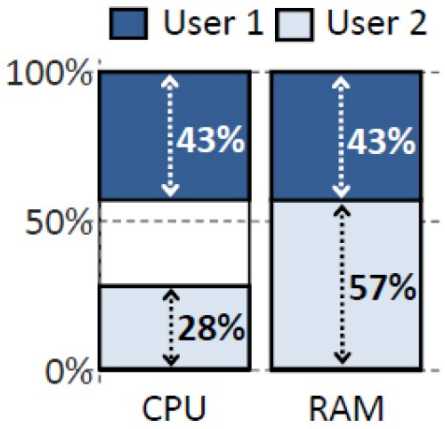
  4x = 6y



  - User 1: x = 1, i.e., <43%CPU, 43%GB> (sum = 86%)
  - User 2: y = 8 i.e., <29%CPU, 57%GB> (sum = 86%)

A first (wrong) solution (2)

- Problem: violates share guarantee
  - User 1 gets less than 50% of both CPU and RAM
  - Better off in a separate cluster with half the resources



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
  - User 1 dominant share is 25%

DRF (2)

- Apply max-min fairness to dominant shares: give every user an equal share of its dominant resource
- 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)

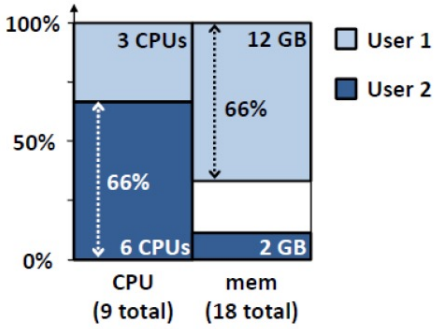
max(x, y)

$x + 3y \leq 9$

$4x + y \leq 18$

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

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



Online DRF

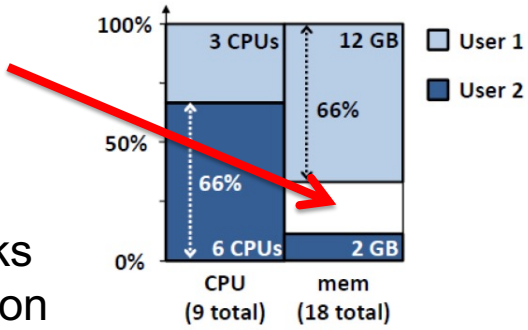
- Whenever there are available resources and tasks to run:

  Choose the framework with the lowest dominant share among all frameworks

Schedule	User A		User B		CPU total alloc.	RAM total alloc.
	res. shares	dom. share	res. shares	dom. share		
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 has 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?)