Imperial College London

LSDS
Large-Scale Data & Systems Group

Orchestration and Scheduling at Cloud Scale

Jana Giceva
jgiceva@doc.ic.ac.uk

Department of Computing
Imperial College London
http://lsds.doc.ic.ac.uk

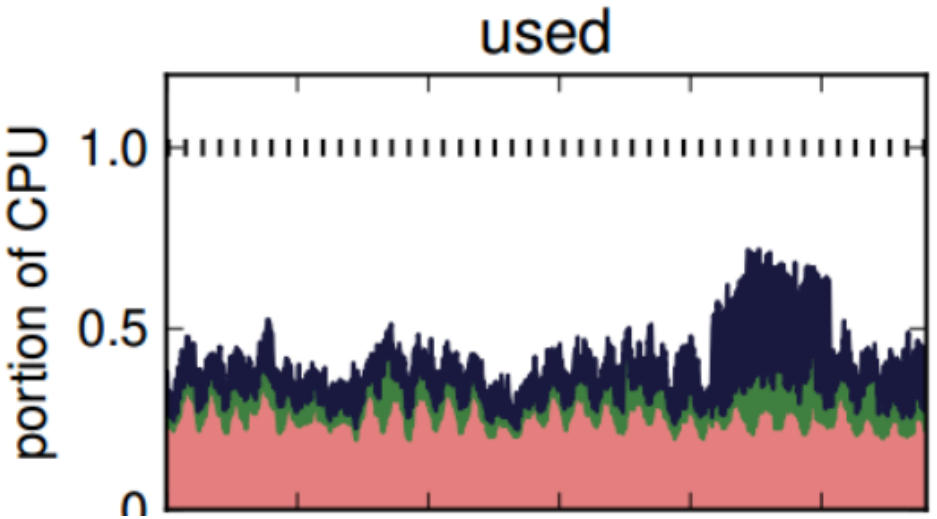
Fall 2018

Understanding the problem

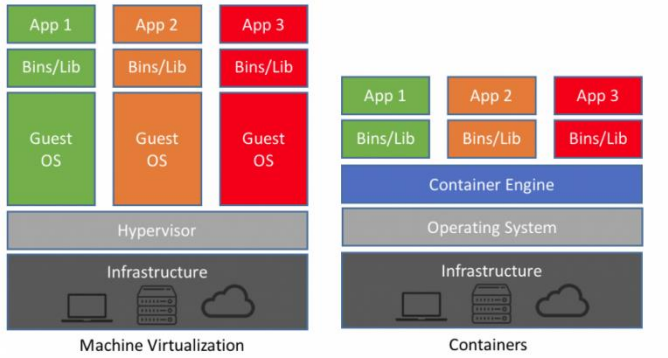
Quick recap

Learned about the internal architecture and scale of datacenters

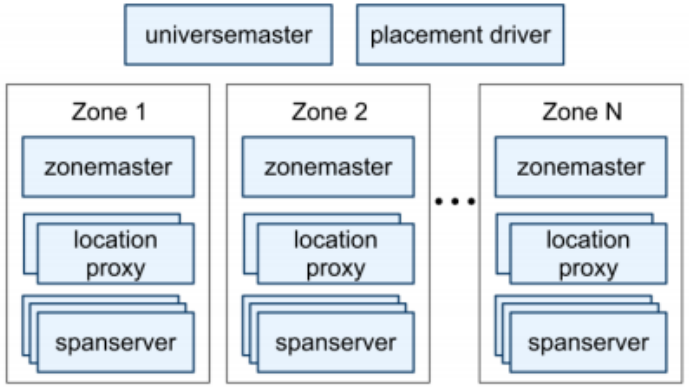
Motivated the need for virtualization



Introduced the concepts of containers



Learned how to design scalable systems



Challenge 1: The need for container orchestration

Managing containers is challenging:

- They are light-weight, flexible and fast.
- But, designed to be short-lived and fragile.
- In the real-world, they fail more often than VMs.

Need for scheduling and orchestration systems for containers:

- Working at data-center scale (large clusters of machines).
- The environment must detect a container failure and replace it immediately.
- If a container is mis-behaving, have the power to kill and re-deploy.
- Ensure that containers are spread reasonably across machines and manage overall hardware resources for the cluster.

Challenge 1: The need for container orchestration

At cloud-scale, the challenge shifts from configuration management to orchestration, scheduling and isolation.

1. *Isolation* – A failure of one computing unit cannot take down another.

2. *Orchestration* – Resources should be reasonably well-balanced geographically to distribute load.

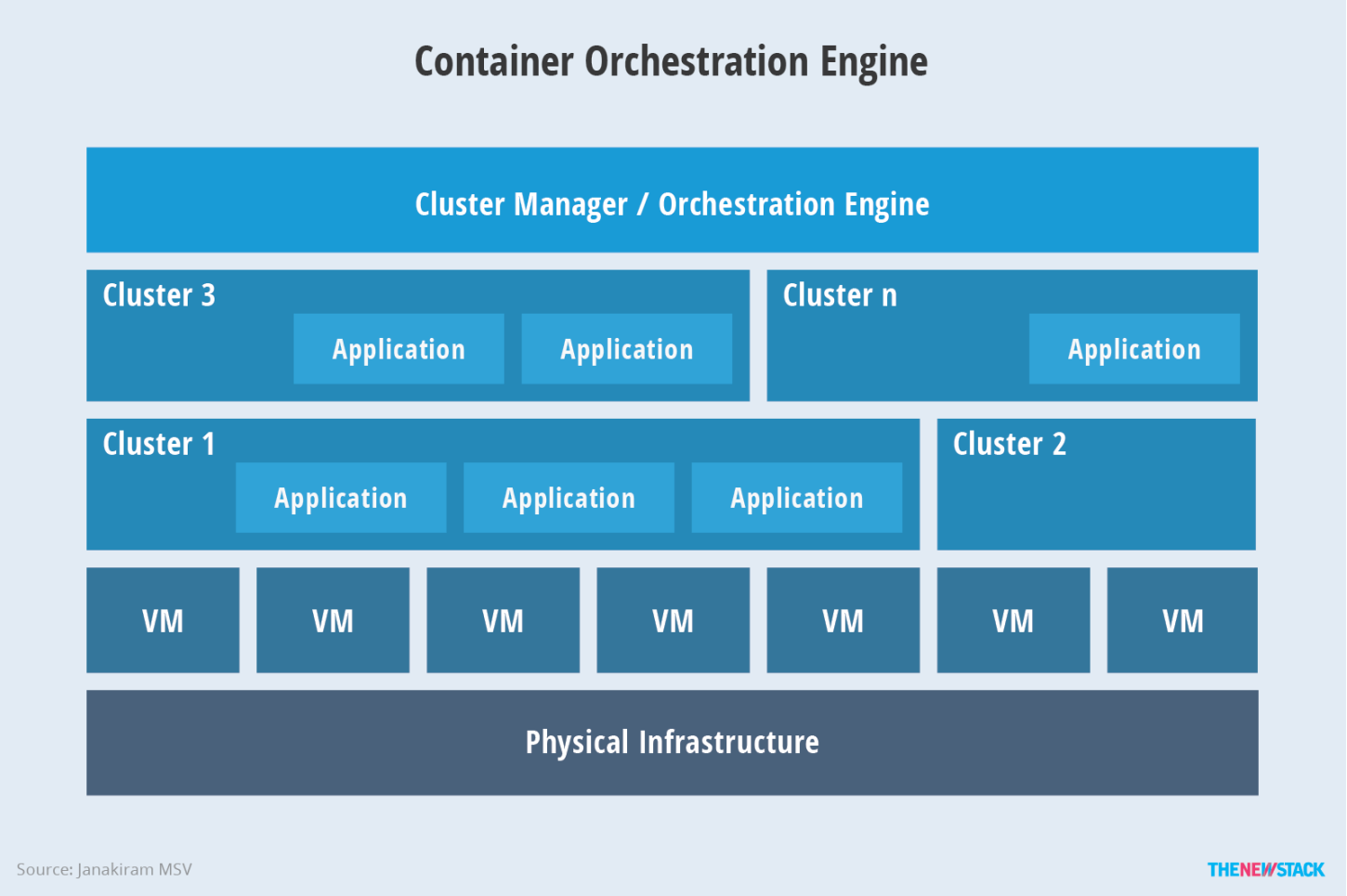
3. *Scheduling* – Need to detect and replace failures near instantaneously.

Cluster management systems

Container orchestration

Container orchestration provides an efficient model for

- Packaging
- Deployment
- Isolation
- Service Discovery
- Scaling
- Load balancing and
- Rolling upgrades.



src: https://thenewstack.io/kubernetes-an-overview/

Applications vs. Services

In the world of containers, one thinks in terms of services instead of applications.

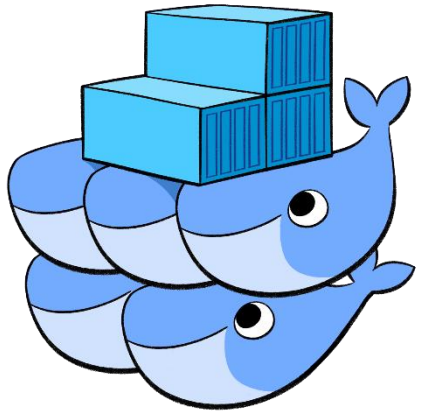
A service is a process that:

- Is designed to do a small number of things (often just one)
- Has no user interface and is invoked solely via an API

An orchestration engine does not manage a fleet of applications,
but a cluster of services.

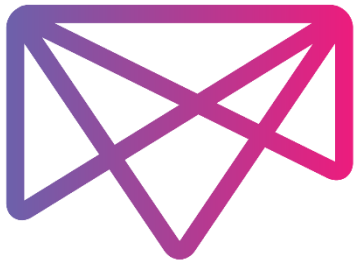
Orchestration Engines

Docker Swarm



https://docs.docker.com/engine/swarm/

Mesosphere DC/OS



https://mesosphere.com/

Kubernetes

https://kubernetes.io/

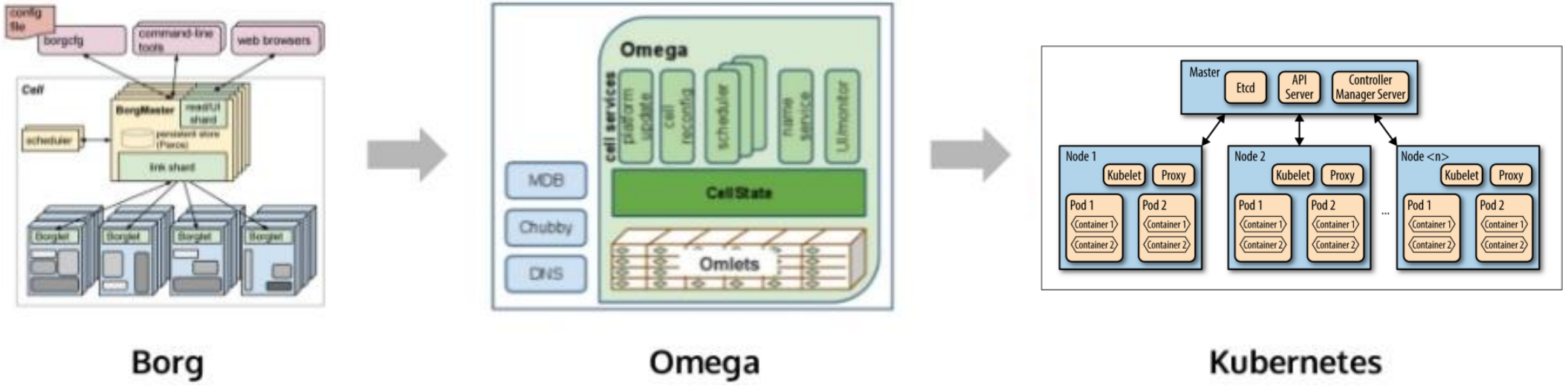
All with the goal to simplify the administration of containers and micro-services and maximize the utilization of data-center resources.

Kubernetes

Kubernetes beginnings

At their scale, Google had to create a set of tools for the orchestration and scheduling of software to handle isolation, load balancing, and placement.

Their first container cluster manager was called Borg [1], and it schedules and launches approximately 7000 containers a second every day.



Borg Omega Kubernetes

[1] Large-scale cluster management at Google with Borg (EuroSys'13)

Kubernetes

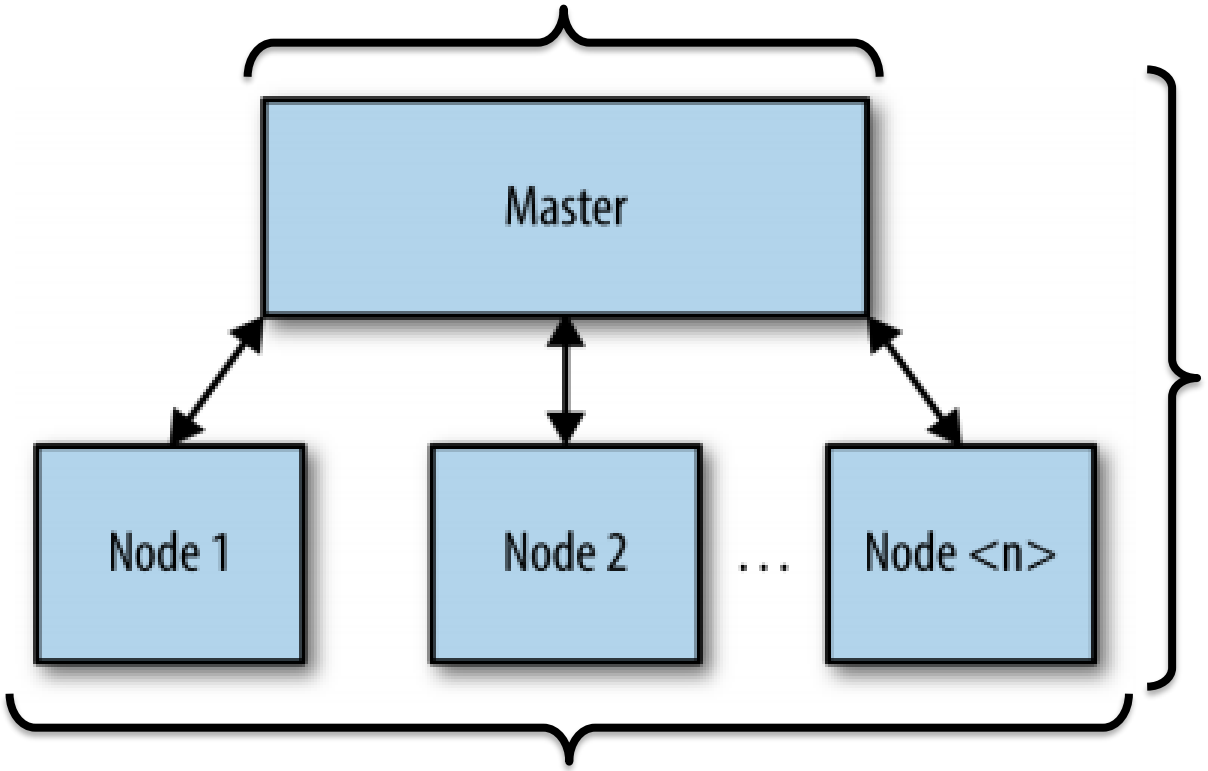
Container management, scheduling and service discovery.

It is composed of control processes that continuously drive the current state towards the desired state.

- API driven application management.
- Agents monitor endpoints for state changes (at real time).
- Controllers enforce the desired state.
- Labels identify resources (nodes, applications, services).

Simplified view of the basic Kubernetes layout

- Masters run special coordinating software that schedules containers on the nodes.

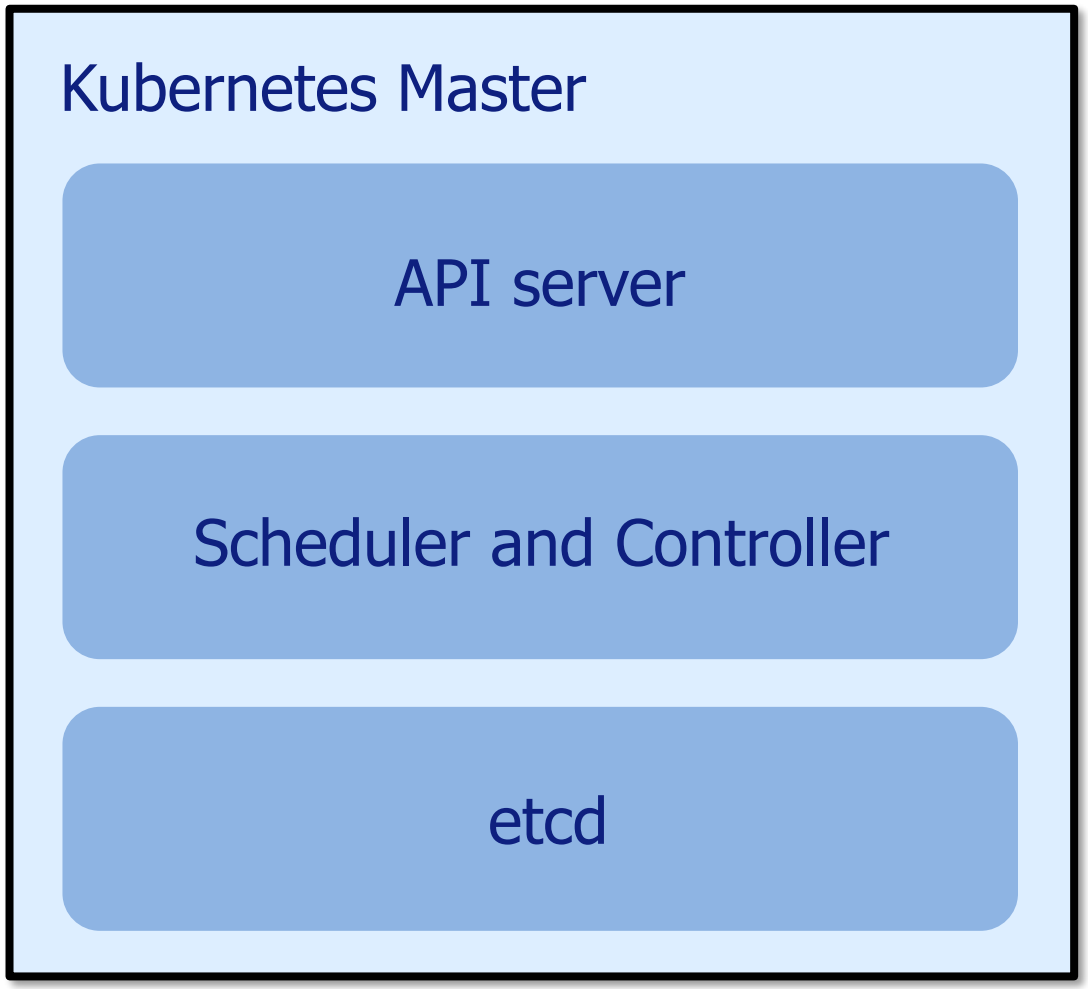


Bunch of machines sit networked together in many datacenters.

Collection of masters and nodes are known as clusters.

- Worker machines are called nodes (minions).
- Each machine hosts 1+ Docker container.

The Master



1. API server

- Nearly all components of the master and nodes accomplish their tasks by making API calls.
- These calls are handled by the API server running on the master.

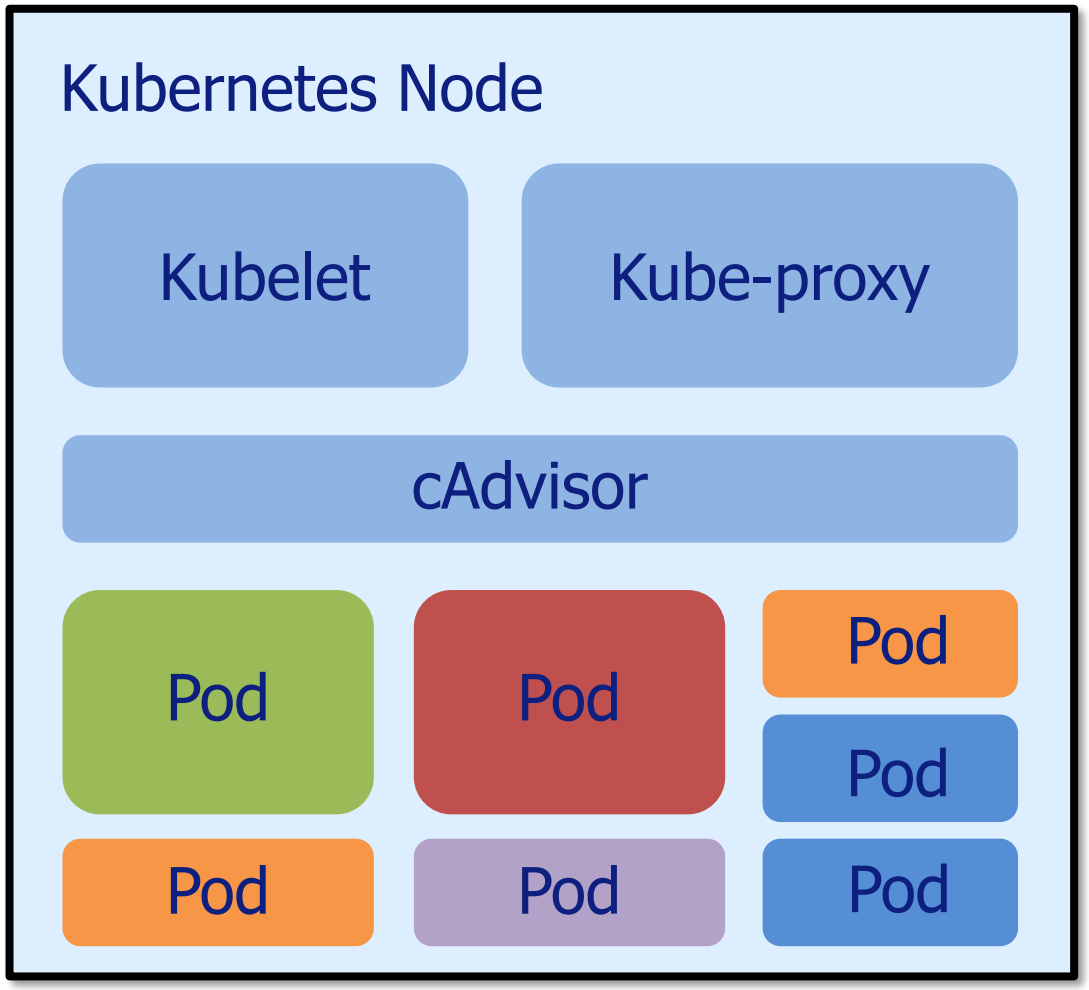
2. Scheduler and Controller Manager

- Processes that schedule containers (i.e., pods) onto target nodes.
- Make sure that the correct number of these things are always running.

3. etcd

- Responsible to keep and replicate the current configuration and run state of the cluster.
- Implemented as light-weight distributed KV store.

The nodes



1. Kubelet

- special background process (daemon)
- execute commands from the master to create, destroy, and monitor containers on that host.

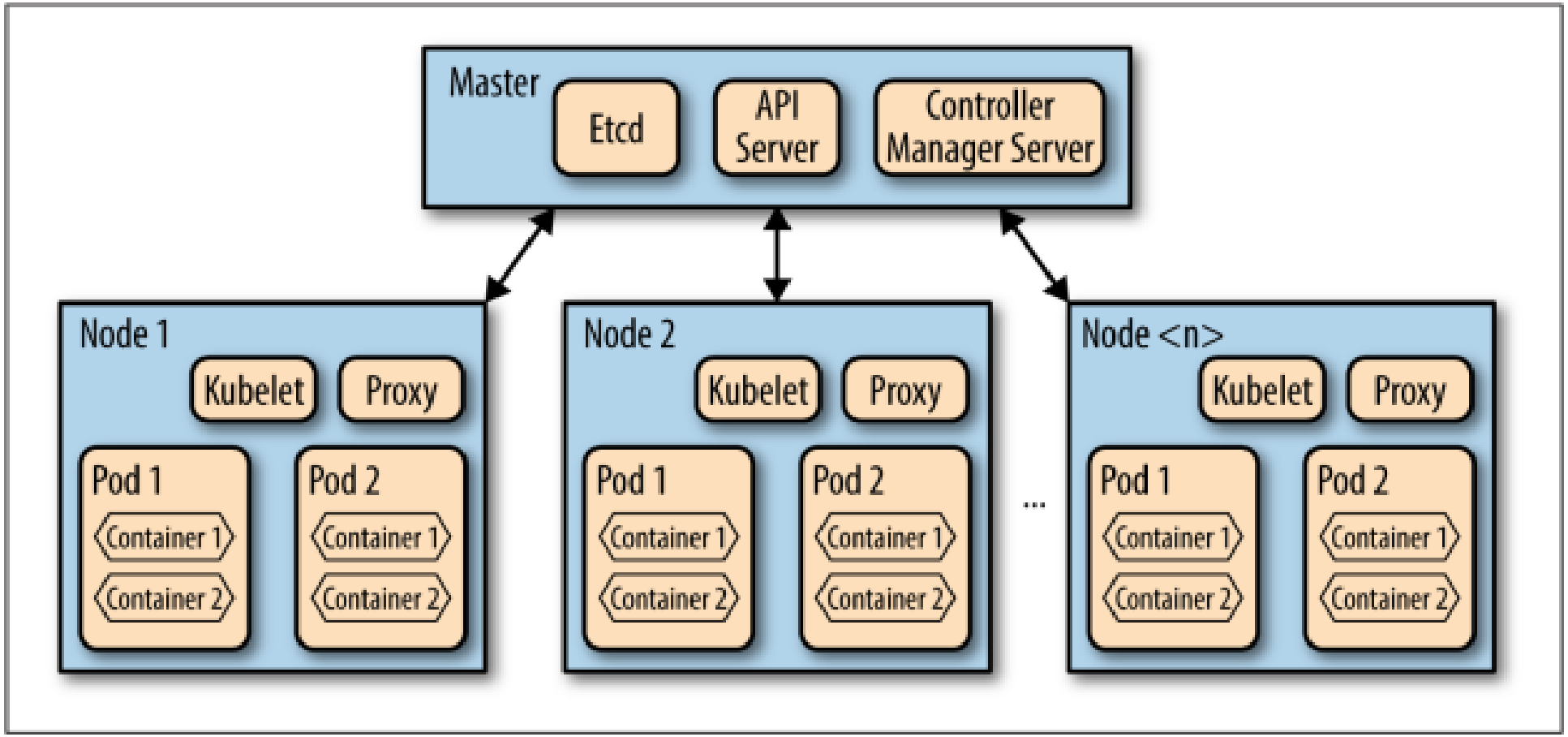
2. Kube-proxy

- simple network proxy to separate the IP address of target container from the name of the service it provides.

3. cAdvisor (optional)

- is a special daemon that collects, aggregates, processes, and exports information about the running containers.

Expanded Kubernetes Layout



src: Kubernetes – Scheduling the future at Cloud Scale, David K. Rensin

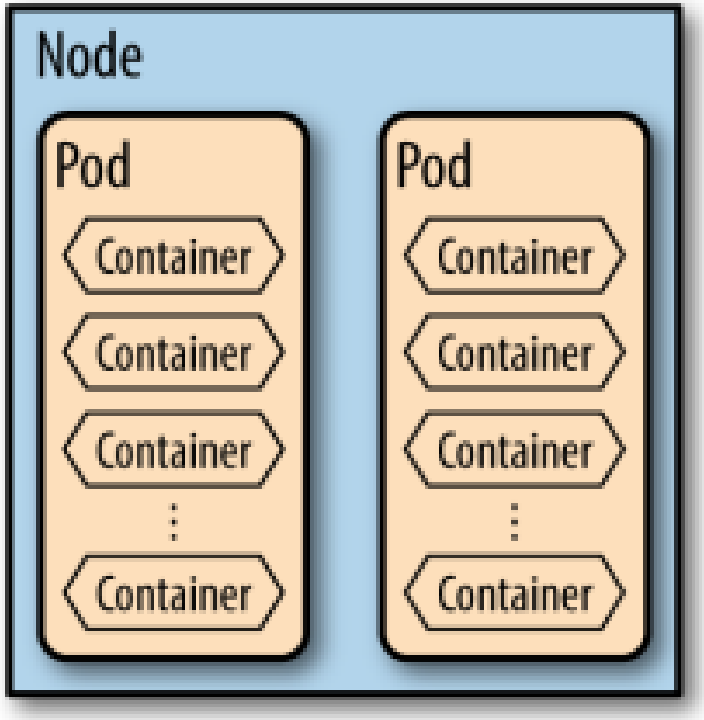
Pods

A pod is a collection of containers that are bundled and scheduled together because they share a common resource – usually a file system or IP address.

Pod serves as Kubernetes' core unit of management.

Pods make up the difference between containerization and virtualization by making it possible to run multiple dependent processes together.

At runtime, pods can be scaled by creating replica sets.



Why not just run multiple programs in a single container?

1. Transparency

- 1+ process in a container – **you** are responsible for monitoring and managing the resources each uses.
- By separating logical units of work into separate containers – **Kubernetes** can manage for you
- Makes things easier to debug and fix.

2. Deployment and Maintenance

- Individual containers can be rebuilt and redeployed by you whenever you make a software change.
- This decoupling of deployment dependencies will make your development and testing faster.
- It also makes it super easy to rollback in case there is a problem.

3. Efficiency

- The infrastructure takes on more responsibility, so the containers can be lighter-weight.

Other important concepts

Replica set ensures that a specified number of pod replicas are running at a time.

- Makes sure that a pod or a set of pods are always up and available.
- If one pod dies, the Replica set will create a pod to make up for it.

Deployment controller provides declarative updates for Pods and Replica Sets.

- Describe a desired state in a Deployment object (Pod(s) and Replica set), and the controller changes the actual state to the desired at a controlled rate.
- Used with a service tier for scaling horizontally or ensuring availability.

Service is an abstraction which defines a logical set of Pods and a policy by which to access them – sometimes called a micro-service.

- When you create a pod, you do not know where it is. It may even be killed by someone.
- Service provides an endpoint that can be addressed by name and be connected to pods using *label* selectors. They are the *external face* of your container workloads.
- Kubernetes sets up a DNS server for the cluster that watches for new services and allows them to be addressed by name.

How it all works?

1. Deploy Kubernetes resources using a Yaml file with kubectl command. It sends Post request to the API server.
2. The API server stores the data into etcd (distributed KV store).
3. Other resources (e.g., Controller Manager, Scheduler, etc.) observe the changed state in the API server.
4. If the Deployment controller detects a deployment object, it creates a ReplicaSet object on the API server.
5. The ReplicaSet controller detects the change, and creates Pods objects.
6. The Scheduler, which is in charge of pod resource allocation, commands the kubelet to execute the docker command and create containers.
7. Every worker node has a kube-proxy to control the routing.

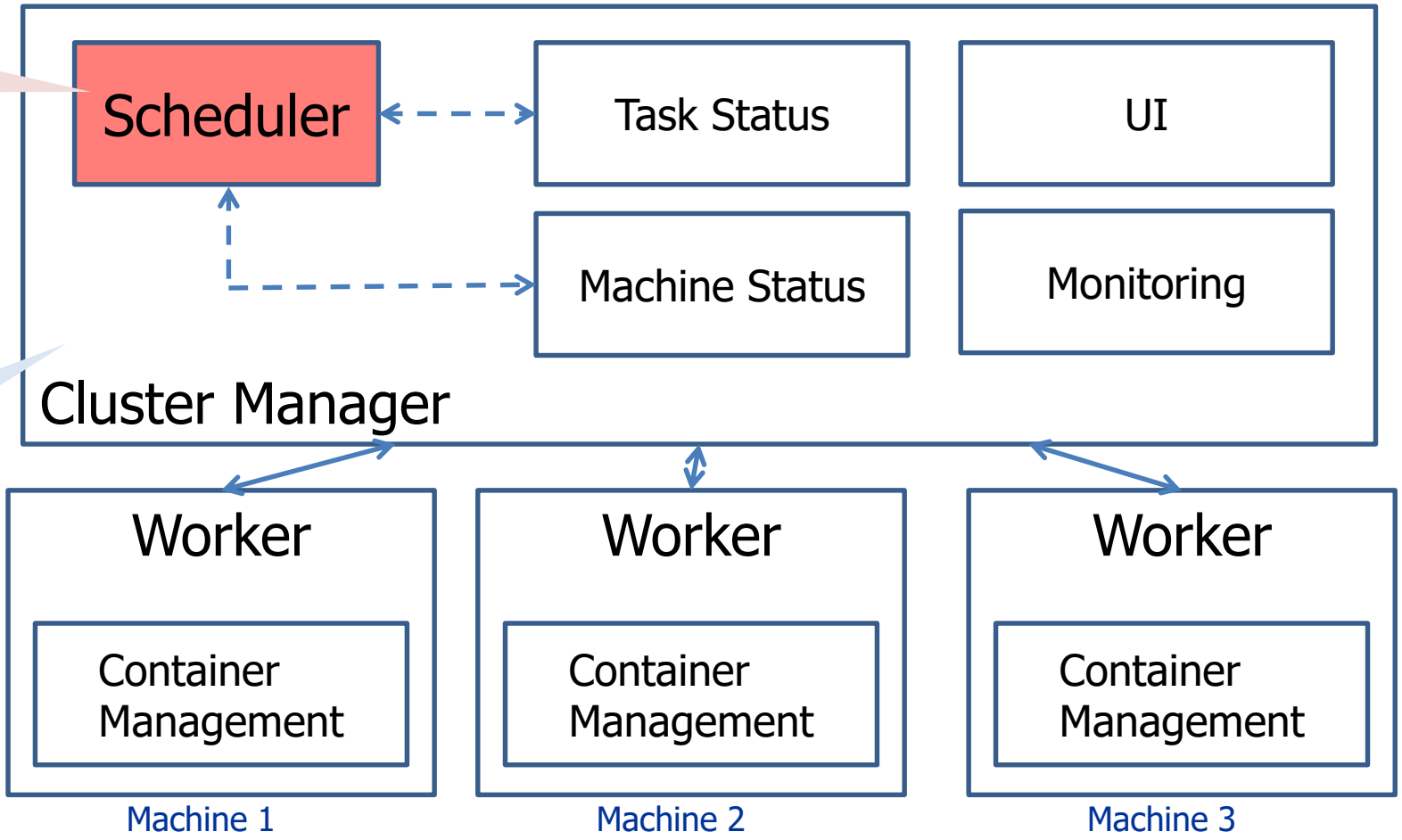
Scheduling and Resource Allocation

Resource Management in Modern Clusters

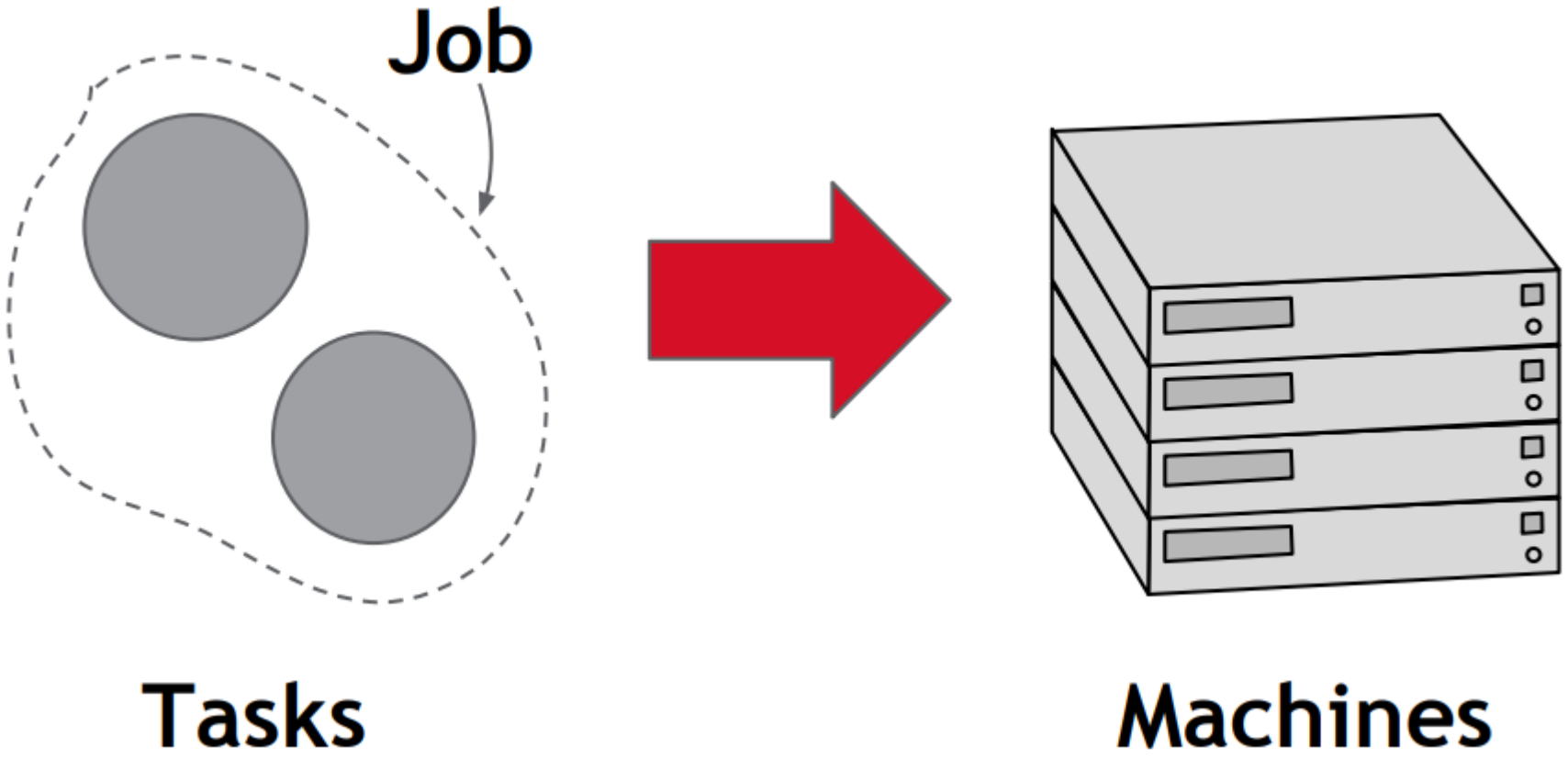
We have a platform to manage many containers over clusters of nodes in a data-center.

How to manage / schedule many scalable systems and other applications at scale?

Goal is to achieve high resource utilization!



What's the role of a scheduler?



src Omega – flexible, scalable schedulers for large compute clusters

Why do we need it?

Workload consolidation in shared clusters.

Goal: better machine resource utilization

In practice, however, there are many challenges:

- Co-location interference
- Noisy neighbors
- Machine heterogeneity
- Overprovisioning of resources

Side effect: poor cluster utilization

- 20% Twitter (Delimitrou et al., ASPLOS 2014)
- 7% Amazon EC2 (Liu et al., DASC 2011)
- 60% Google (Verma et al., EuroSys 2015)

Goals and requirements for Cluster Schedulers

Cluster utilization (Return of Investment)

User-supplied resource constraints

Job completion time

Allocation latency (rapid decision making)

Scalability

Sharing constraints

Various degrees of “fairness”

Challenging scale

Jobs process **gigabytes to petabytes** of data and

issue peaks of **100’000 scheduling requests/second.**

Clusters run up to **170’000 tasks in parallel**, and

each contains **over 20’000 servers.**

Challenge 1:
How to make optimal scheduling decisions at production scale?

Heterogeneous Load

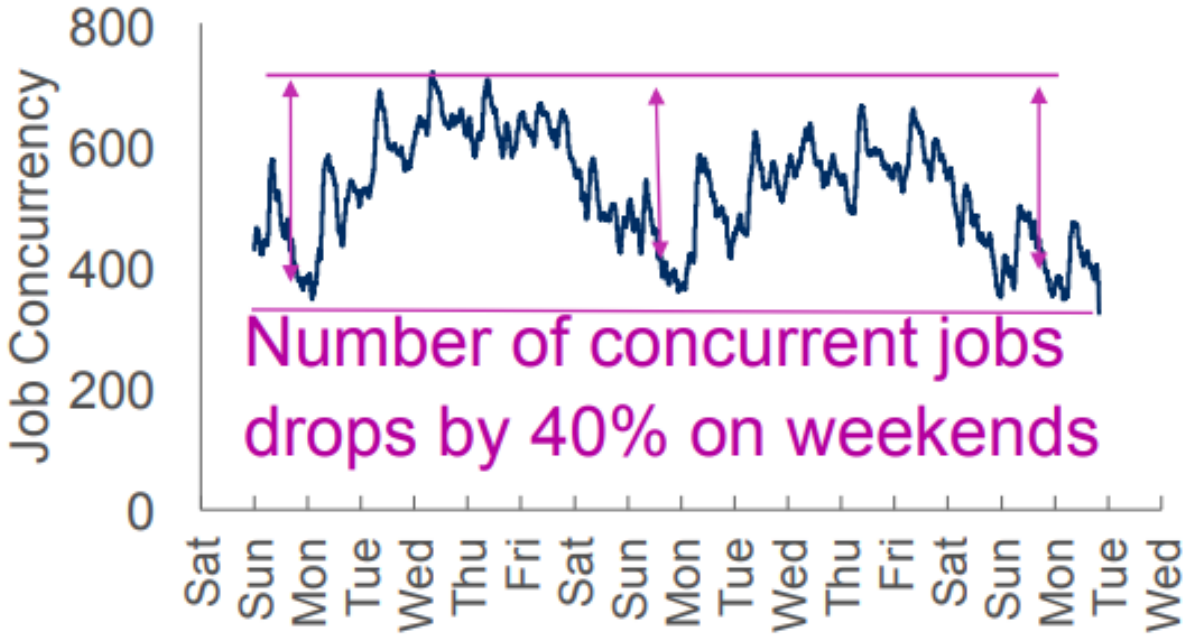
Datacenters run heterogeneous workloads:

- **Duration**: tasks run from seconds to hours
- **Resource-type demand**: tasks can be IO-bound or CPU-bound
- **Resource-amount demand**: tasks can require from 100MB to 10+GB memory
- **Latency-sensitivity:** short tasks are often sensitive to scheduling latency
- **Locality-sensitivity:** long IO-bound tasks are sensitive to locality

Challenge 2:
How to make optimal scheduling decisions for a complex heterogeneous workload?

Maximize resource utilization

We need to **effectively use resources** and **maintain performance guarantees**, but the **workload constantly fluctuates**.



src: Apollo – scalable and coordinated scheduler for cloud-scale computing

Challenge 3:
How maximize utilization while maintaining performance guarantees with a dynamic workload?

Cluster scheduler designs options

Centralized:

- Good task placements
- Sophisticated algorithms
- Examples: Borg, Quincy, Quasar, Firmament

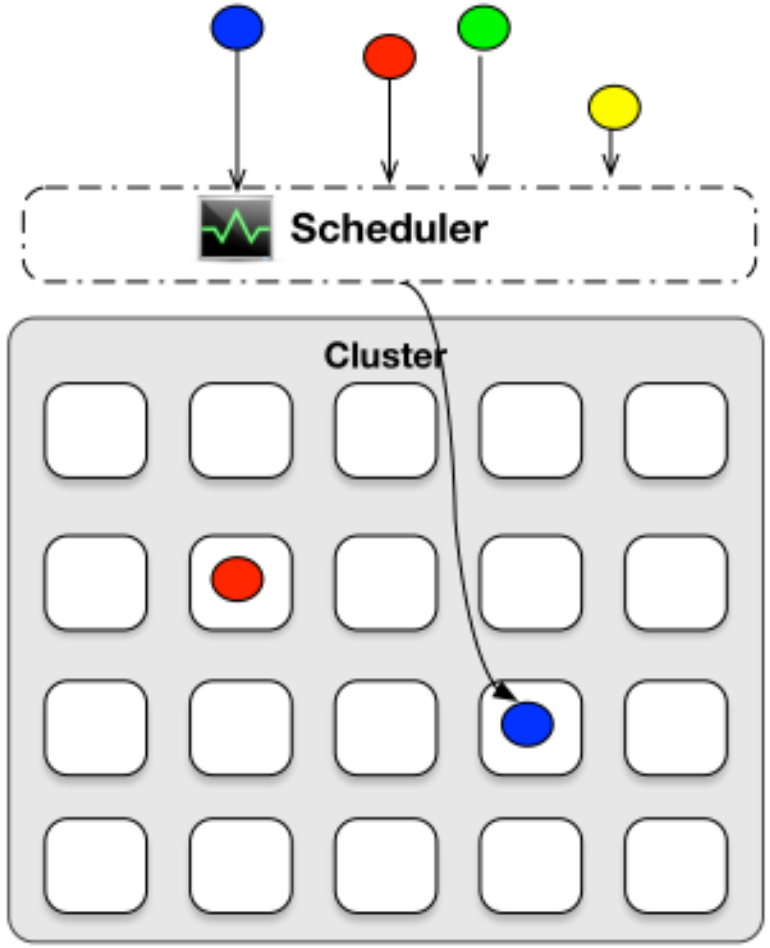
Distributed:

- Low scheduling latency
- Usually rely on simple heuristics
- Examples: Sparrow, Tarcil, Yaq-d

Hybrid:

- Split the workload, provide either
- Two-scheduler design: one for short-lived containers, second for long running jobs
- Examples: Mercury, Hawk, Eagle, Medea

Centralized (Monolithic) Schedulers



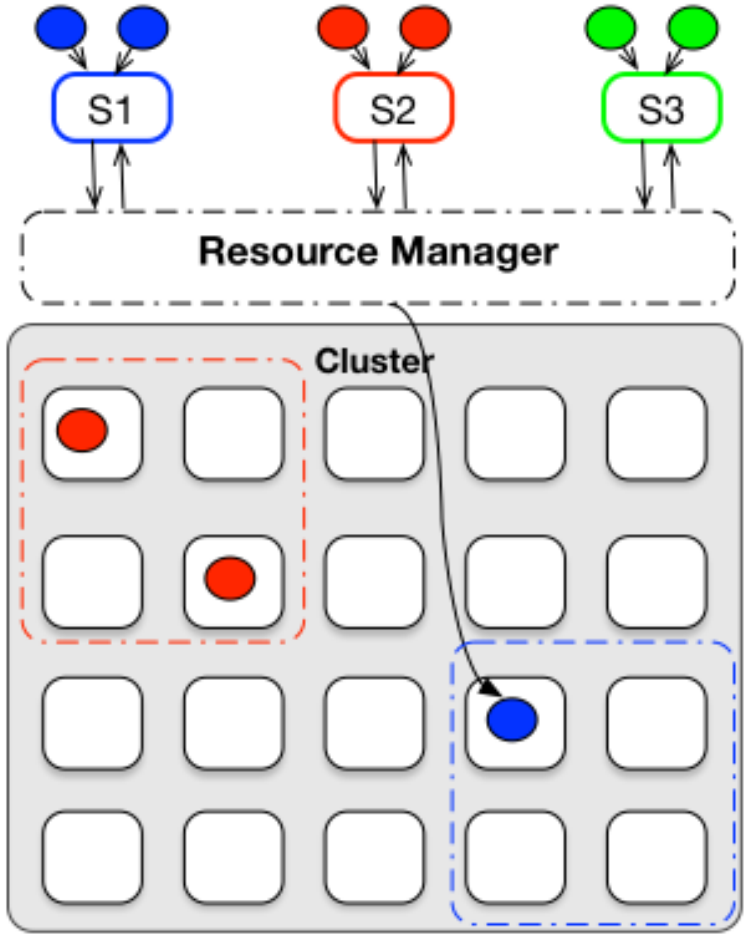
Centralised Scheduler

Monolithic schedulers – entire cluster is scheduled using a centralized component.

- commonly used for HPC workloads
- monolithic code base
- hard to diversify
- difficult to extend
- do not scale well

Example schedulers: Hadoop, Borg, Quincy

Two-level schedulers



Two-level Scheduler

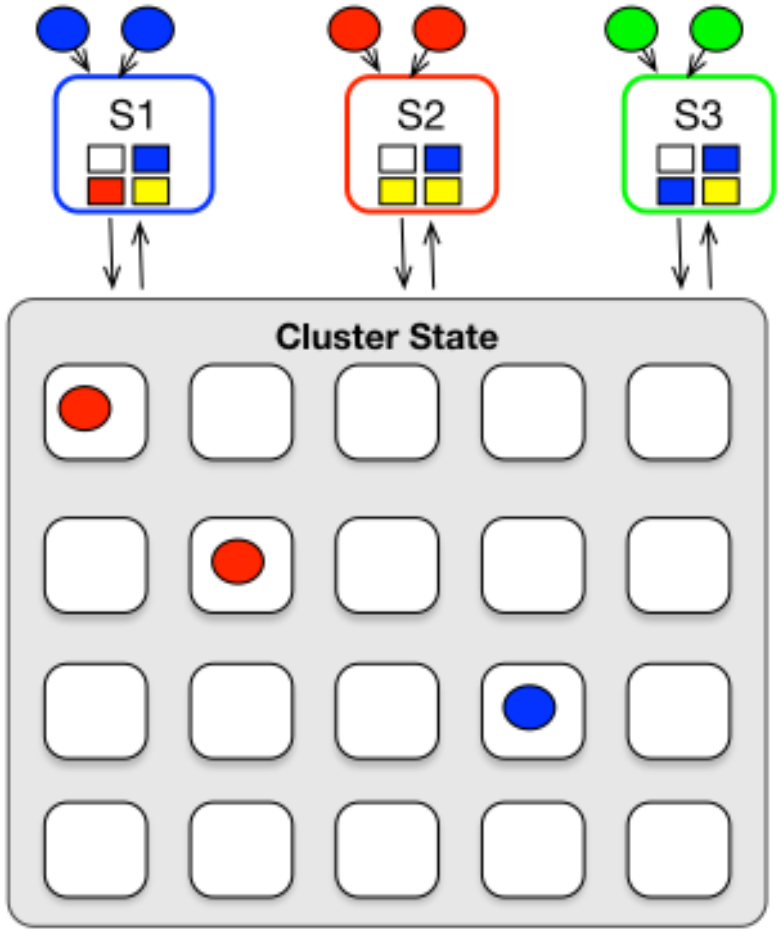
Two-level schedulers use a central allocator that assigns resources to "frameworks" that internally have their own schedulers.

- information hiding
- Hoarding
- cannot support preemption

Example schedulers:

- Mesos, Yarn, Hadoop-on-demand

Shared-state schedulers



Shared-state Scheduler

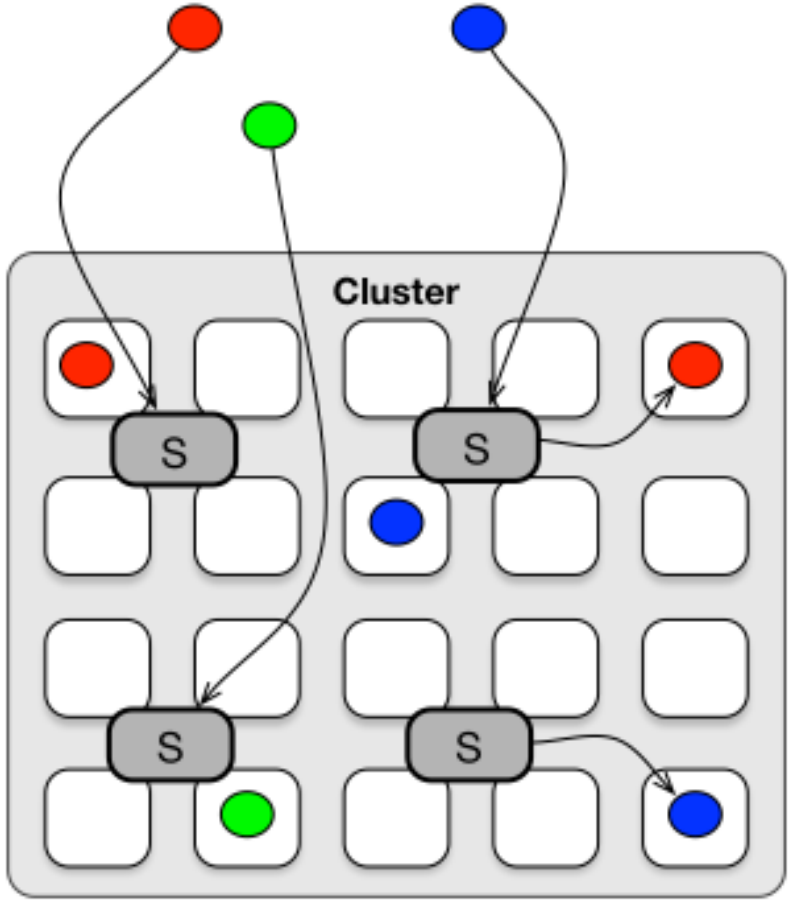
Grant each scheduler full access to the entire cluster state, allow them to compete for resources and resolve conflicts when they arise.

- Good scalability
- Good resource allocation

Example:

- Omega, Apollo

Distributed schedulers



Distributed Scheduler

Distributed schedulers use simple algorithms (heuristics):

- Scale well
- Have good scheduling latencies
- Poor decisions because they do not consider the cluster state

Example:

- Sparrow, Tarcil, Yaq-d

Design decisions

1. Should we partition the scheduling work?

2. How to choose the resources?

3. How to deal with interference and ideally avoid it?

4. At what granularity should we allocate the resources?

5. How do we ensure fairness?

References

1. Kubernetes – Scheduling the Future at Cloud Scale by David K. Rensin
2. https://thenewstack.io/kubernetes-an-overview/
3. Large-scale cluster management at Google with Borg (EuroSys'13)
3. Omega: flexible, scalable schedulers for large compute clusters (EuroSys'13)
4. Apollo: Scalable and Coordinated Scheduler for Cloud-Scale Computing (OSDI'14)
5. Firmament: Fast, Centralized Cluster Scheduling at Scale (OSDI'16)