Desenvolvimento de Aplicações com Arquitetura Baseada em Microservices

Prof. Vinicius Cardoso Garcia
vcg@cin.ufpe.br :: @vinicius3w :: assertlab.com

[IF1007] - Tópicos Avançados em SI 4
https://github.com/vinicius3w/if1007-Microservices

Licença do material

Resources

- There is no textbook required. However, the following are some books that may be recommended:
  - Building Microservices: Designing Fine-Grained Systems
  - Spring Microservices
  - Spring Boot: Acelere o desenvolvimento de microsserviços
  - Microservices for Java Developers A Hands-on Introduction to Frameworks and Containers
  - Migrating to Cloud-Native Application Architectures
  - Continuous Integration
  - Getting started guides from spring.io

Managing Dockerized Microservices with Mesos and Marathon

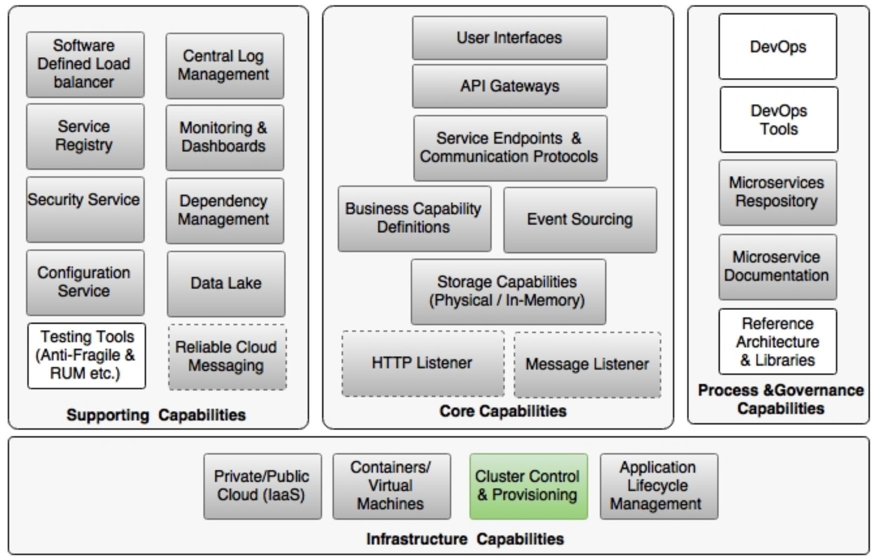
Context

- In an Internet-scale microservices deployment, it is not easy to manage thousands of dockerized microservices
- It is essential to have an infrastructure abstraction layer and a strong cluster control platform to successfully manage Internet-scale microservice deployments
- This lecture will explain the need and use of Mesos and Marathon as an infrastructure abstraction layer and a cluster control system, respectively, to achieve optimized resource usage in a cloud-like environment when deploying microservices at scale

Topics

- The need to have an abstraction layer and cluster control software
- Mesos and Marathon from the context of microservices
- Managing dockerized BrownField Airline's PSS microservices with Mesos and Marathon

Reviewing the microservice capability model


Software Defined Load balancer
Central Log Management
Service Registry
Monitoring & Dashboards
Security Service
Dependency Management
Configuration Service
Data Lake
Testing Tools (Anti-Fragile & RUM etc.)
Reliable Cloud Messaging
Supporting Capabilities
User Interfaces
API Gateways
Service Endpoints & Communication Protocols
Business Capability Definitions
Event Sourcing
Storage Capabilities (Physical / In-Memory)
HTTP Listener
Message Listener
Core Capabilities
DevOps
DevOps Tools
Microservices Respository
Microservice Documentation
Reference Architecture & Libraries
Process &Governance Capabilities
Private/Public Cloud (IaaS)
Containers/ Virtual Machines
Cluster Control & Provisioning
Application Lifecycle Management
Infrastructure Capabilities

AssertLab

In this lecture, we will explore the Cluster Control & Provisioning microservices capability from the microservice capability model discussed in lecture 4, Applying Microservices Concepts.

The missing pieces

- Docker helped package the JVM runtime and OS parameters along with the application
- There is no special consideration required when moving dockerized microservices from one environment to another
- The REST APIs provided by Docker have simplified the life cycle manager's interaction with the target machine in starting and stopping artifacts

The missing pieces

- In a large-scale deployment, with hundreds and thousands of Docker containers, we need to ensure that Docker containers run with their own resource constraints, such as memory, CPU, and so on
- There may be rules set for Docker deployments, such as replicated copies of the container should not be run on the same machine
- A mechanism needs to be in place to optimally use the server infrastructure to avoid incurring extra cost

The missing pieces

- There are organizations that deal with billions of containers, managing them manually is next to impossible
- In the context of large-scale Docker deployments, some of the key questions to be answered are
    1. How do we manage thousands of containers?
    2. How do we monitor them?
    3. How do we apply rules and constraints when deploying artifacts?
    4. How do we ensure that we utilize containers properly to gain resource efficiency?
    5. How do we ensure that at least a certain number of minimal instances are running at any point in time?
    6. How do we ensure dependent services are up and running?
    7. How do we do rolling upgrades and graceful migrations?
    8. How do we roll back faulty deployments?

The missing pieces

- All these questions point to the need to have a solution to address two key capabilities

  I. A cluster abstraction layer that provides a uniform abstraction over many physical or virtual machines

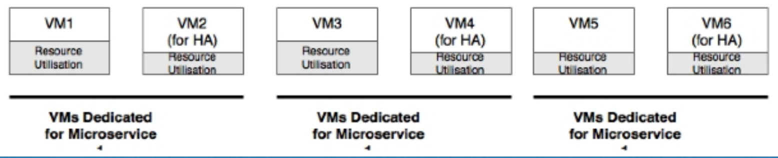
  II. A cluster control and init system to manage deployments intelligently on top of the cluster abstraction

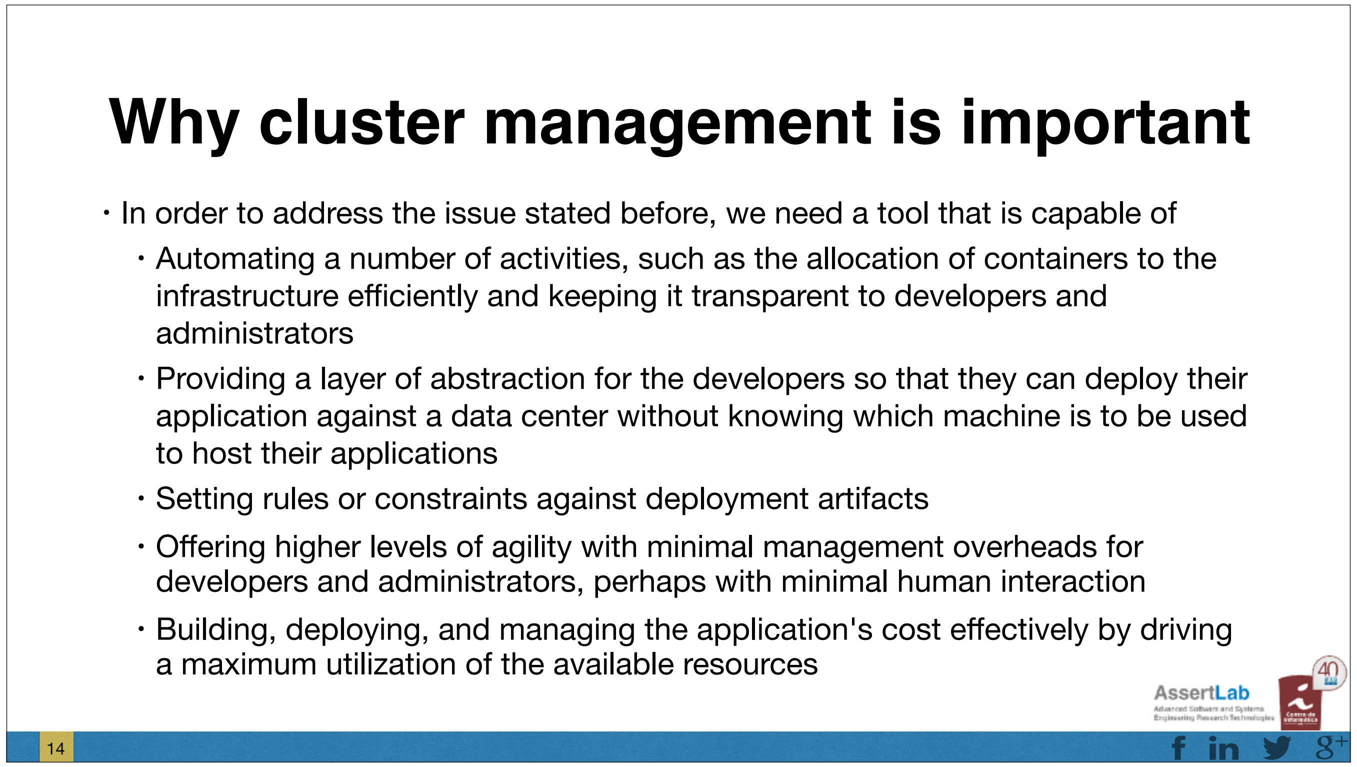
The missing pieces

- The life cycle manager is ideally placed to deal with these situations
- One can add enough intelligence to the life cycle manager to solve these issues
- However, before attempting to modify the life cycle manager, it is important to understand the role of cluster management solutions a bit more.

Why cluster management is important

- As microservices break applications into different micro-applications, many developers request more server nodes for deployment
- In order to manage microservices properly, developers tend to deploy one microservice per VM, which further drives down the resource utilization
- In many cases, this results in an overallocation of CPUs and memory
- In many deployments, the high-availability requirements of microservices force engineers to add more and more service instances for redundancy
- In general, microservice deployment requires more infrastructure compared to monolithic application deployments



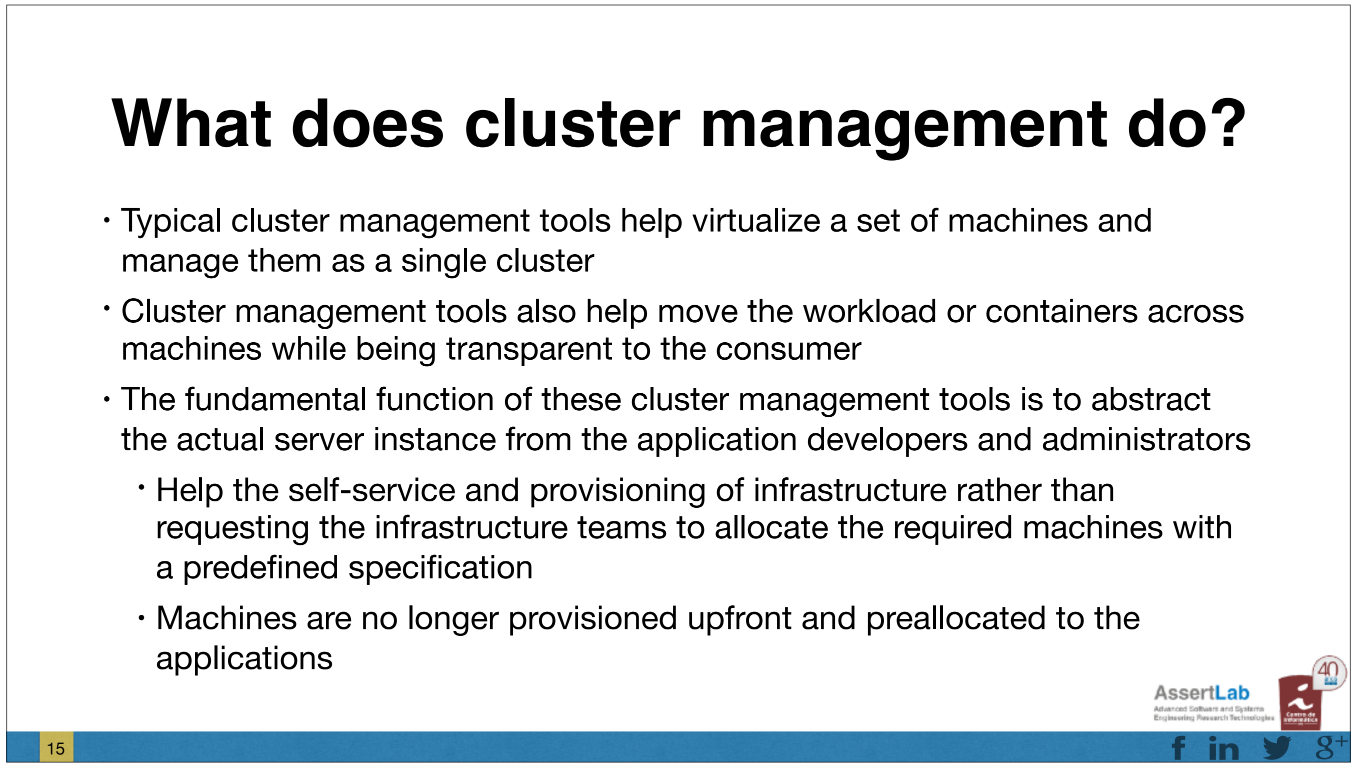


Why cluster management is important

- In order to address the issue stated before, we need a tool that is capable of
  - Automating a number of activities, such as the allocation of containers to the infrastructure efficiently and keeping it transparent to developers and administrators
  - Providing a layer of abstraction for the developers so that they can deploy their application against a data center without knowing which machine is to be used to host their applications
  - Setting rules or constraints against deployment artifacts
  - Offering higher levels of agility with minimal management overheads for developers and administrators, perhaps with minimal human interaction
  - Building, deploying, and managing the application's cost effectively by driving a maximum utilization of the available resources

AssertLab
Advanced Software and Systems Engineering Research Technologies

Containers solve an important issue in this context. Any tool that we select with these capabilities can handle containers in a uniform way, irrespective of the underlying microservice technologies



What does cluster management do?

- Typical cluster management tools help virtualize a set of machines and manage them as a single cluster
- Cluster management tools also help move the workload or containers across machines while being transparent to the consumer
- The fundamental function of these cluster management tools is to abstract the actual server instance from the application developers and administrators
  - Help the self-service and provisioning of infrastructure rather than requesting the infrastructure teams to allocate the required machines with a predefined specification
  - Machines are no longer provisioned upfront and preallocated to the applications

Technology evangelists and practitioners use different terminologies, such as cluster orchestration, cluster management, data center virtualization, container schedulers, or container life cycle management, container orchestration, data center operating system, and so on.
Some of the cluster management tools also help virtualize data centers across many heterogeneous machines or even across data centers, and create an elastic, private cloud-like infrastructure

Key capabilities of cluster management software

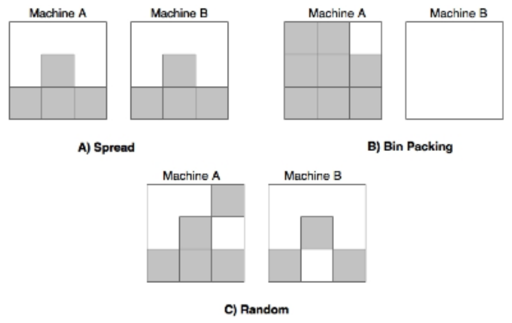
- **Cluster** management: It manages a cluster of VMs and physical machines as a single large machine
- **Deployments**: It handles the automatic deployment of applications and containers with a large set of machines
- **Scalability**: It handles the automatic and manual scalability of application instances as and when required, with optimized utilization as the primary goal
- **Health**: It manages the health of the cluster, nodes, and applications
- **Infrastructure abstraction**: It abstracts the developers from the actual machine on which the applications are deployed

Key capabilities of cluster management software

- **Resource optimization**: The inherent behavior of these tools is to allocate container workloads across a set of available machines in an efficient way
- **Resource allocation**: It allocates servers based on resource availability and the constraints set by application developers
- **Service availability**: It ensures that the services are up and running somewhere in the cluster
- **Agility**: These tools are capable of quickly allocating workloads to the available resources or moving the workload across machines if there is change in resource requirements
- **Isolation**: Some of these tools provide resource isolation out of the box. Hence, even if the application is not containerized, resource isolation can be still achieved

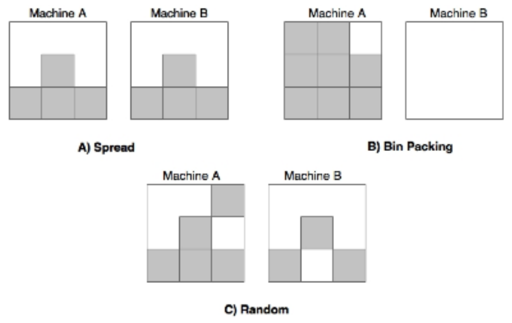
Key capabilities of cluster management software

- A variety of algorithms are used for resource allocation, ranging from simple algorithms to complex algorithms, with machine learning and artificial intelligence
- The common algorithms used are random, bin packing, and spread
- Constraints set against applications will override the default algorithms based on resource availability

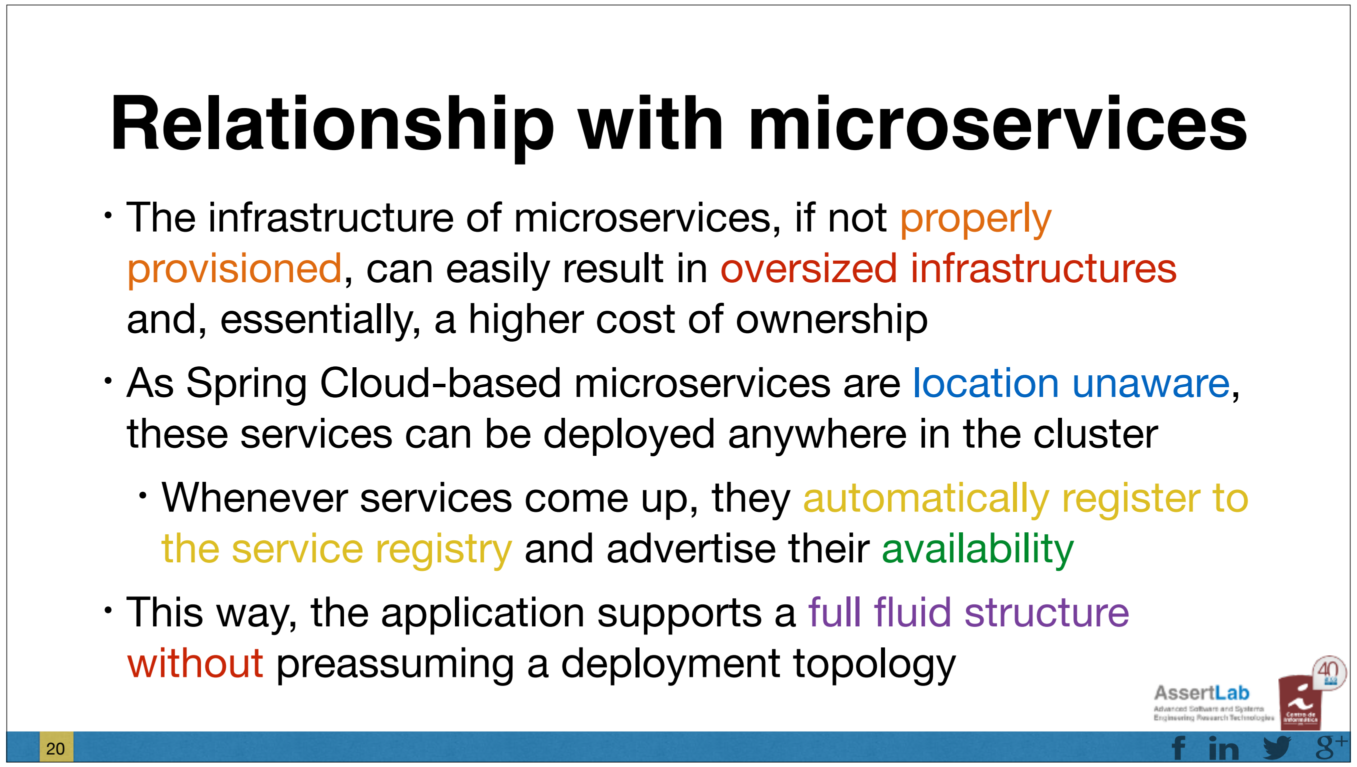


Key capabilities of cluster management software

- **Spread**: This algorithm performs the allocation of workload equally across the available machines
- **Bin packing**: This algorithm tries to fill in data machine by machine and ensures the maximum utilization of machines. Bin packing is especially good when using cloud services in a pay-as-you-use style
- **Random**: This algorithm randomly chooses machines and deploys containers on randomly selected machines



There is a possibility of using cognitive computing algorithms such as machine learning and collaborative filtering to improve efficiency. Techniques such as oversubscription allow a better utilization of resources by allocating underutilized resources for high-priority tasks—for example, revenue-generating services for best-effort tasks such as analytics, video, image processing, and so on.



Relationship with microservices

- The infrastructure of microservices, if not properly provisioned, can easily result in oversized infrastructures and, essentially, a higher cost of ownership
- As Spring Cloud-based microservices are location unaware, these services can be deployed anywhere in the cluster
  - Whenever services come up, they automatically register to the service registry and advertise their availability
- This way, the application supports a full fluid structure without preassuming a deployment topology

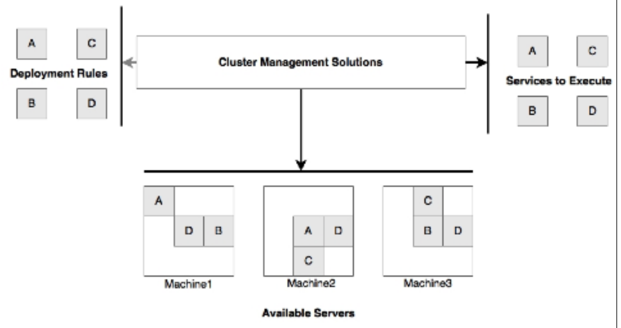
As discussed in the previous sections, a cloud-like environment with a cluster management tool is essential to realize cost benefits when dealing with large-scale microservices.

Relationship with virtualization

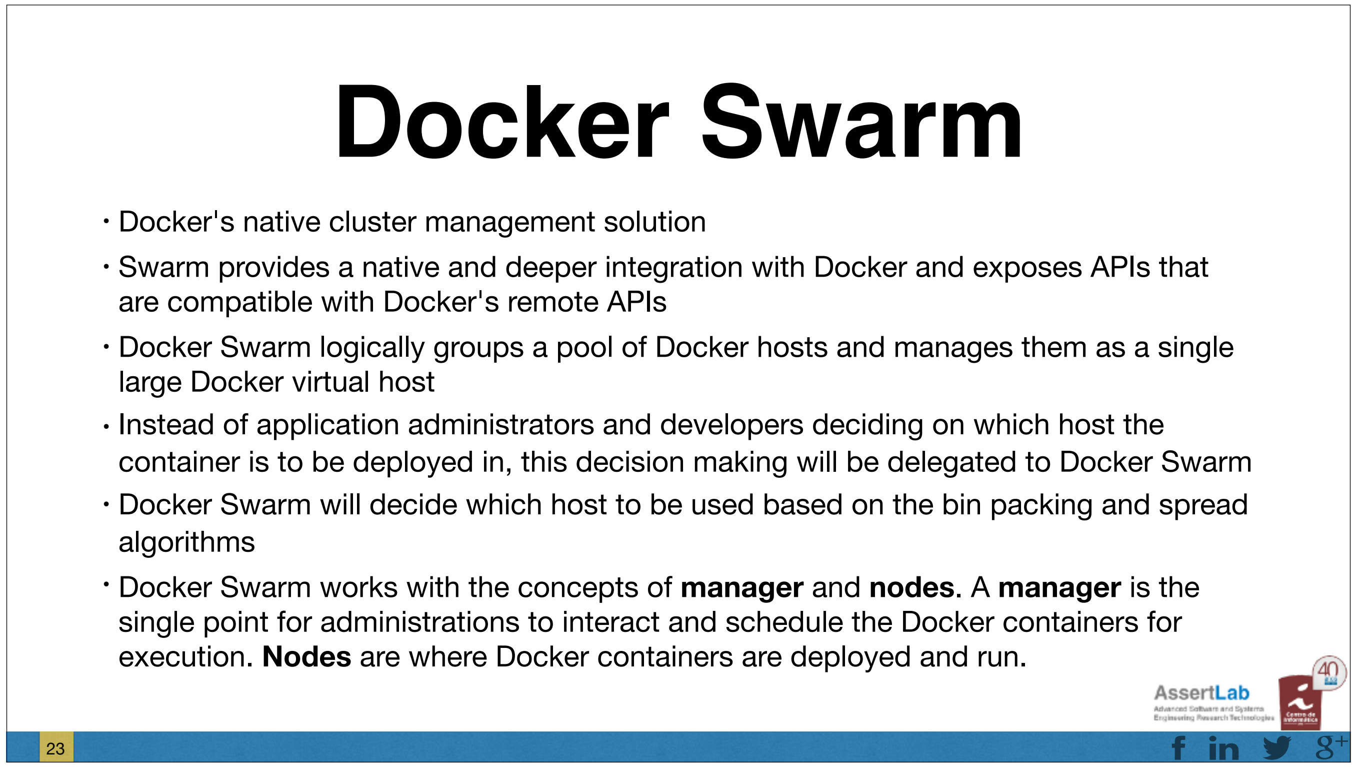
- Cluster management solutions are different from server virtualization solutions in many aspects
- Cluster management solutions run on top of VMs or physical machines as an application component

Cluster management solutions

- There are many cluster management software tools available
- It is unfair to do an apple-to-apple comparison between them
- Even though there are no one-to-one components, there are many areas of overlap in capabilities between them
- In many situations, organizations use a combination of one or more of these tools to fulfill their requirements



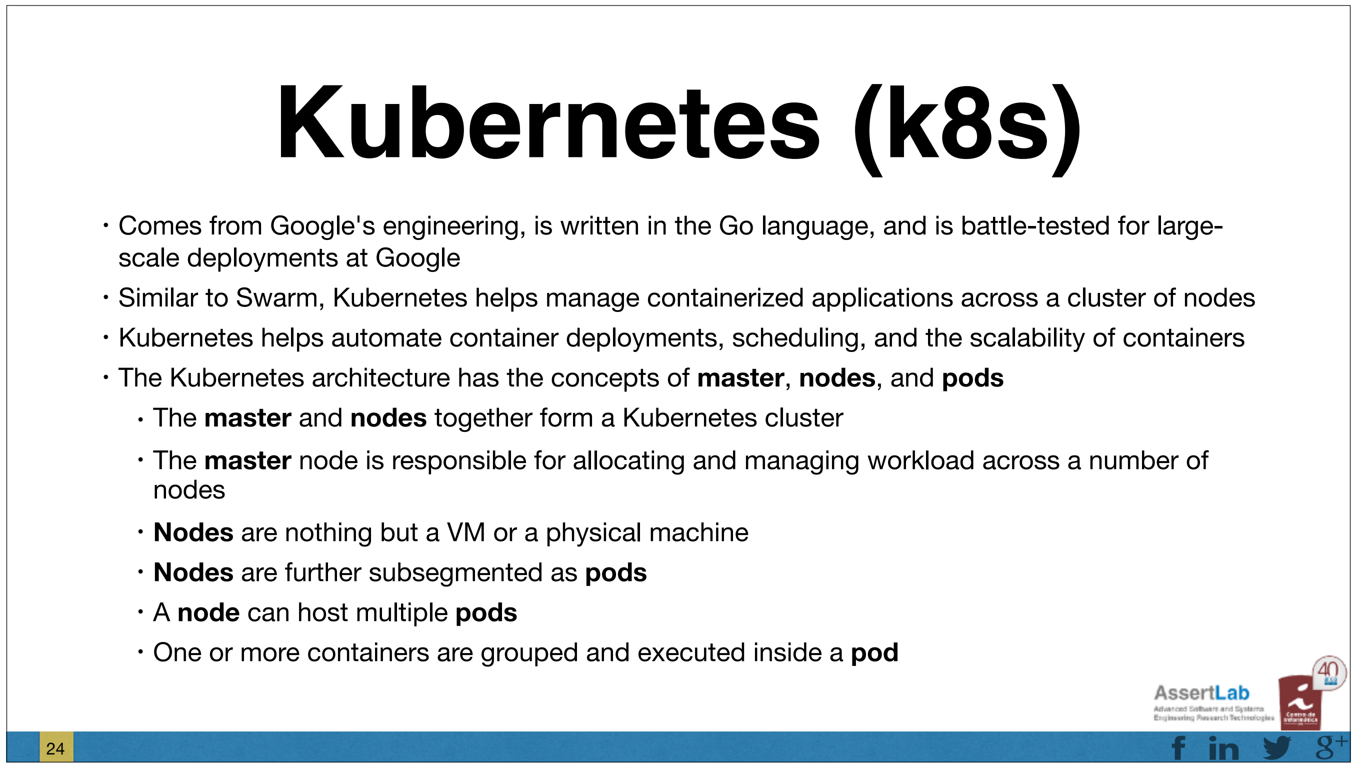
AssertLab
Advanced Software and Systems Engineering Research Technologies



Docker Swarm

- Docker's native cluster management solution
- Swarm provides a native and deeper integration with Docker and exposes APIs that are compatible with Docker's remote APIs
- Docker Swarm logically groups a pool of Docker hosts and manages them as a single large Docker virtual host
- Instead of application administrators and developers deciding on which host the container is to be deployed in, this decision making will be delegated to Docker Swarm
- Docker Swarm will decide which host to be used based on the bin packing and spread algorithms
- Docker Swarm works with the concepts of **manager** and **nodes**. A **manager** is the single point for administrations to interact and schedule the Docker containers for execution. **Nodes** are where Docker containers are deployed and run.

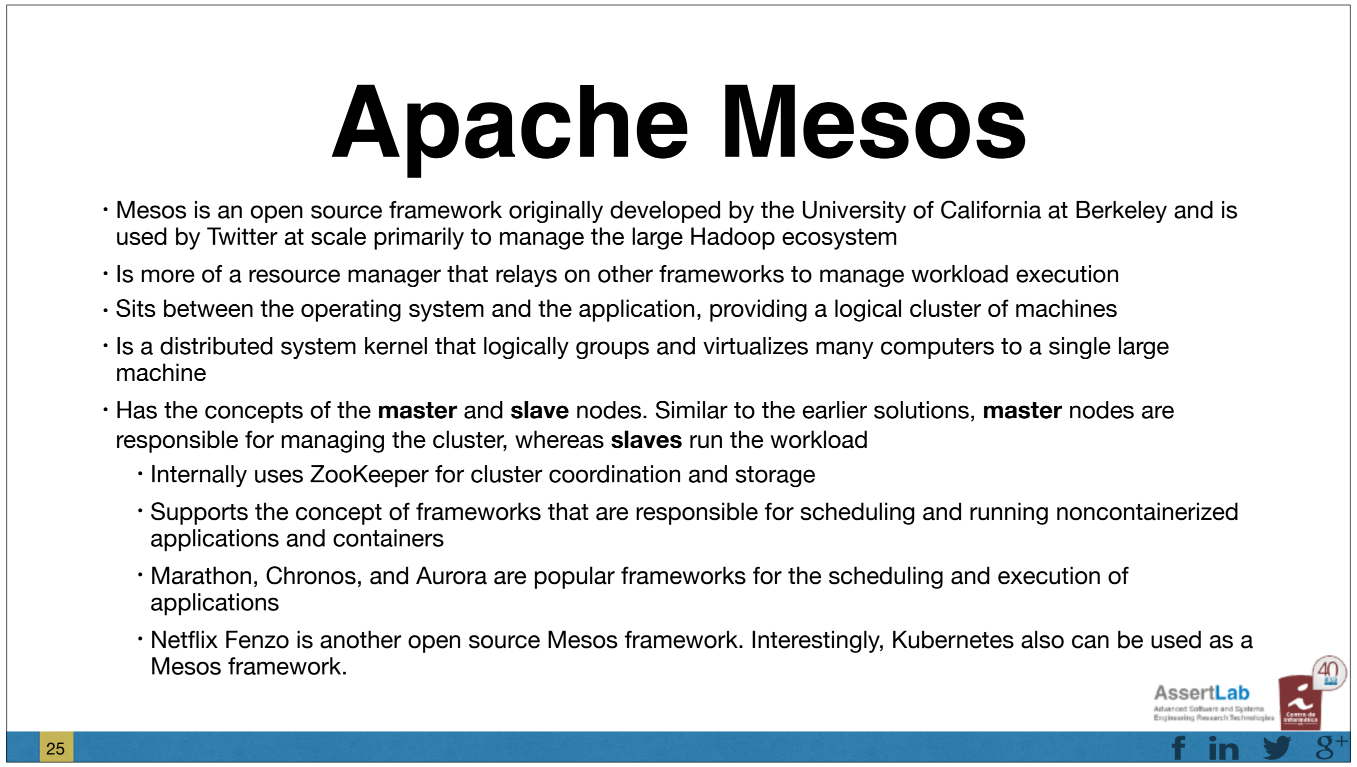
As Docker Swarm is based on Docker's remote APIs, its learning curve for those already using Docker is narrower compared to any other container orchestration tools. However, Docker Swarm is a relatively new product on the market, and it only supports Docker containers.
Docker Swarm works with the concepts of manager and nodes. A manager is the single point for administrations to interact and schedule the Docker containers for execution. Nodes are where Docker containers are deployed and run.



Kubernetes (k8s)

- Comes from Google's engineering, is written in the Go language, and is battle-tested for large-scale deployments at Google
- Similar to Swarm, Kubernetes helps manage containerized applications across a cluster of nodes
- Kubernetes helps automate container deployments, scheduling, and the scalability of containers
- The Kubernetes architecture has the concepts of **master**, **nodes**, and **pods**
  - The **master** and **nodes** together form a Kubernetes cluster
  - The **master** node is responsible for allocating and managing workload across a number of nodes
  - **Nodes** are nothing but a VM or a physical machine
  - **Nodes** are further subsegmented as **pods**
  - A **node** can host multiple **pods**
  - One or more containers are grouped and executed inside a **pod**

Kubernetes also supports the concept of labels as key-value pairs to query and find containers. Labels are user-defined parameters to tag certain types of nodes that execute a common type of workloads, such as frontend web servers. The services deployed on a cluster get a single IP/DNS to access the service



Apache Mesos

- Mesos is an open source framework originally developed by the University of California at Berkeley and is used by Twitter at scale primarily to manage the large Hadoop ecosystem
- Is more of a resource manager that relays on other frameworks to manage workload execution
- Sits between the operating system and the application, providing a logical cluster of machines
- Is a distributed system kernel that logically groups and virtualizes many computers to a single large machine
- Has the concepts of the **master** and **slave** nodes. Similar to the earlier solutions, **master** nodes are responsible for managing the cluster, whereas **slaves** run the workload
  - Internally uses ZooKeeper for cluster coordination and storage
  - Supports the concept of frameworks that are responsible for scheduling and running noncontainerized applications and containers
  - Marathon, Chronos, and Aurora are popular frameworks for the scheduling and execution of applications
  - Netflix Fenzo is another open source Mesos framework. Interestingly, Kubernetes also can be used as a Mesos framework.

Mesos is capable of grouping a number of heterogeneous resources to a uniform resource cluster on which applications can be deployed. For these reasons, Mesos is also known as a tool to build a private cloud in a data center.
Marathon supports the Docker container as well as noncontainerized applications. Spring Boot can be directly configured in Marathon. Marathon provides a number of capabilities out of the box, such as supporting application dependencies, grouping applications to scale and upgrade services, starting and shutting down healthy and unhealthy instances, rolling out promotes, rolling back failed promotes, and so on.
Mesosphere offers commercial support for Mesos and Marathon as part of its DCOS platform

Nomad

- Nomad is a cluster management system that abstracts lower-level machine details and their locations
- Nomad has a simpler architecture compared to the other solutions explored earlier
- Nomad has the concept of **servers**, in which all **jobs** are managed
- One **server** acts as the **leader**, and others act as **followers**
- Nomad has the concept of **tasks**, which is the smallest unit of work
  - **Tasks** are grouped into **task groups**
  - A **task group** has **tasks** that are to be executed in the same location
  - One or more **task groups** or **tasks** are managed as **jobs**
- Nomad supports many workloads, including Docker, out of the box
- Nomad also supports deployments across data centers and is region and data center aware

Fleet

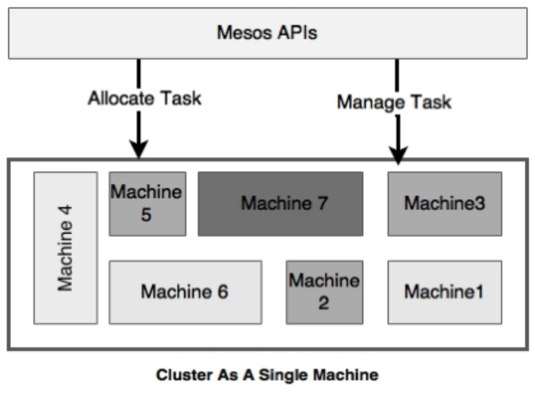
- Fleet is a cluster management system from CoreOS
- It runs on a lower level and works on top of systemd
- Fleet can manage application dependencies and make sure that all the required services are running somewhere in the cluster
- If a service fails, it restarts the service on another host. Affinity and constraint rules are possible to supply when allocating resources
- Fleet has the concepts of **engine** and **agents**
  - There is only one **engine** at any point in the cluster with multiple **agents**
  - Tasks are submitted to the engine and agent run these tasks on a cluster machine

Cluster management with Mesos and Marathon

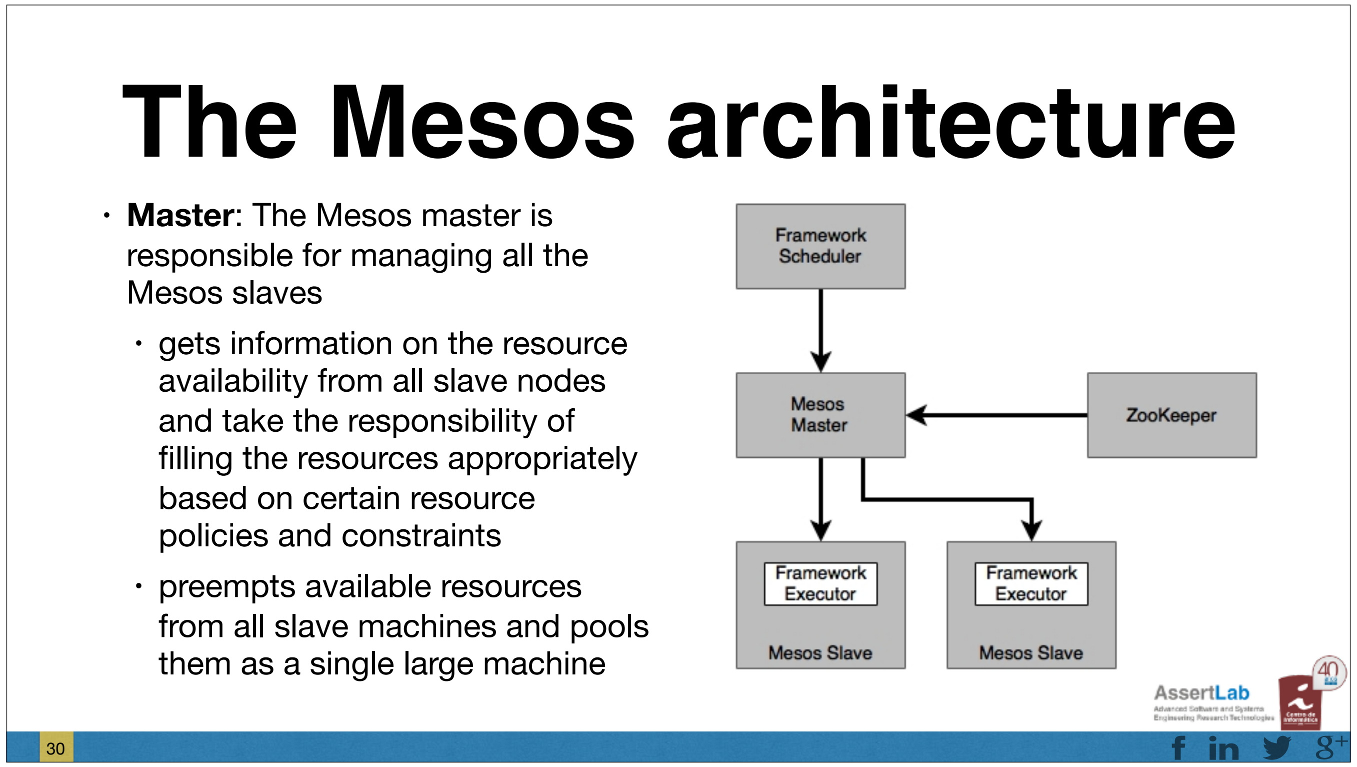
- Many organizations choose Kubernetes or Mesos with a framework such as Marathon
  - In most cases, Docker is used as a default containerization method to package and deploy workloads
- For the rest of this lecture, we will show how Mesos works with Marathon to provide the required cluster management capability
- Mesos is used by many organizations, including Twitter, Airbnb, Apple, eBay, Netflix, PayPal, Uber, Yelp, and many others

Diving deep into Mesos

- Mesos can be treated as a data center kernel
- In order to run multiple tasks on one node, Mesos uses resource isolation concepts
- Mesos relies on the Linux kernel's **cgroups** to achieve resource isolation similar to the container approach
    - It also supports containerized isolation using Docker
- Mesos supports both batch workload as well as the OLTP kind of workloads

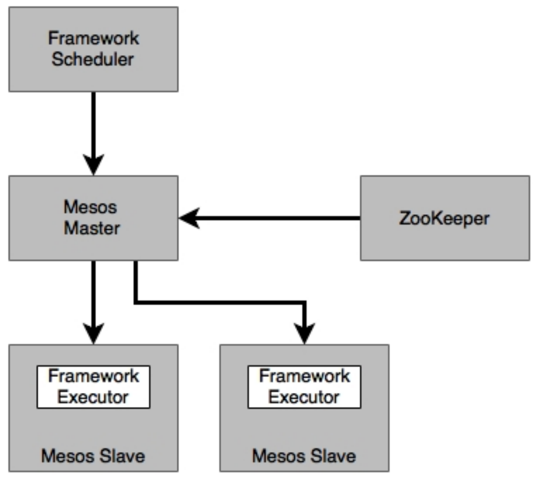


DCOS is the commercial version of Mesos supported by Mesosphere.
Mesos is an open source top-level Apache project under the Apache license. Mesos abstracts lower-level computing resources such as CPU, memory, and storage from lower-level physical or virtual machines.



The Mesos architecture

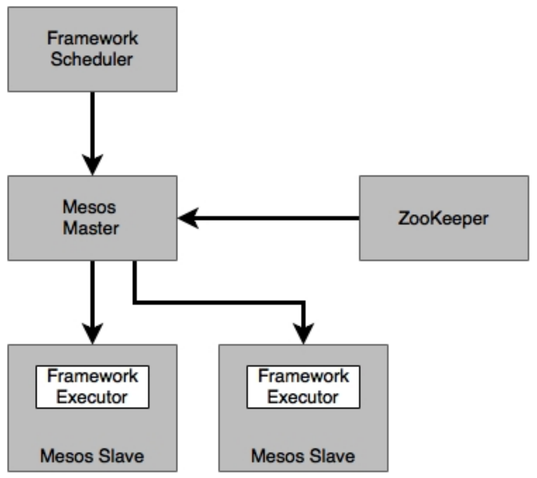
- **Master**: The Mesos master is responsible for managing all the Mesos slaves
  - gets information on the resource availability from all slave nodes and take the responsibility of filling the resources appropriately based on certain resource policies and constraints
  - preempts available resources from all slave machines and pools them as a single large machine



For high availability, the Mesos master is supported by the Mesos master's standby components. Even if the master is not available, the existing tasks can still be executed. However, new tasks cannot be scheduled in the absence of a master node. The master standby nodes are nodes that wait for the failure of the active master and take over the master's role in the case of a failure. It uses ZooKeeper for the master leader election. A minimum quorum requirement must be met for leader election.

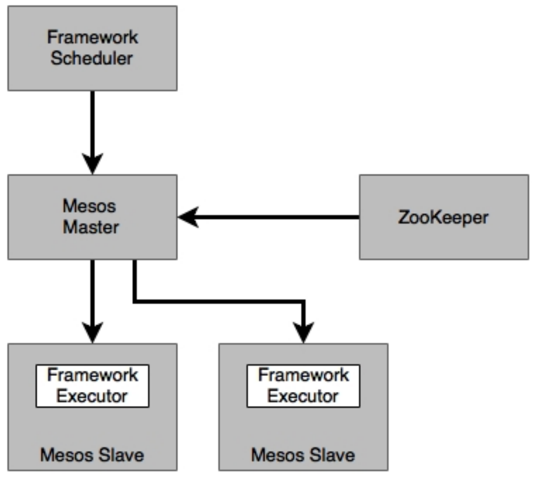
The Mesos architecture

- **Slave**: Mesos slaves are responsible for hosting task execution frameworks
  - Tasks are executed on the slave nodes
  - Mesos slaves can be started with attributes as key-value pairs, such as data center = X
  - This is used for constraint evaluations when deploying workloads
  - Slave machines share resource availability with the Mesos master



The Mesos architecture

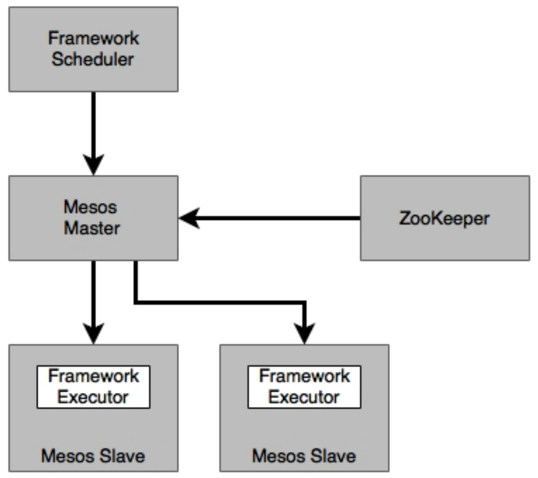
- **ZooKeeper**: ZooKeeper is a centralized coordination server used in Mesos to coordinate activities across the Mesos cluster
  - Mesos uses ZooKeeper for leader election in case of a Mesos master failure

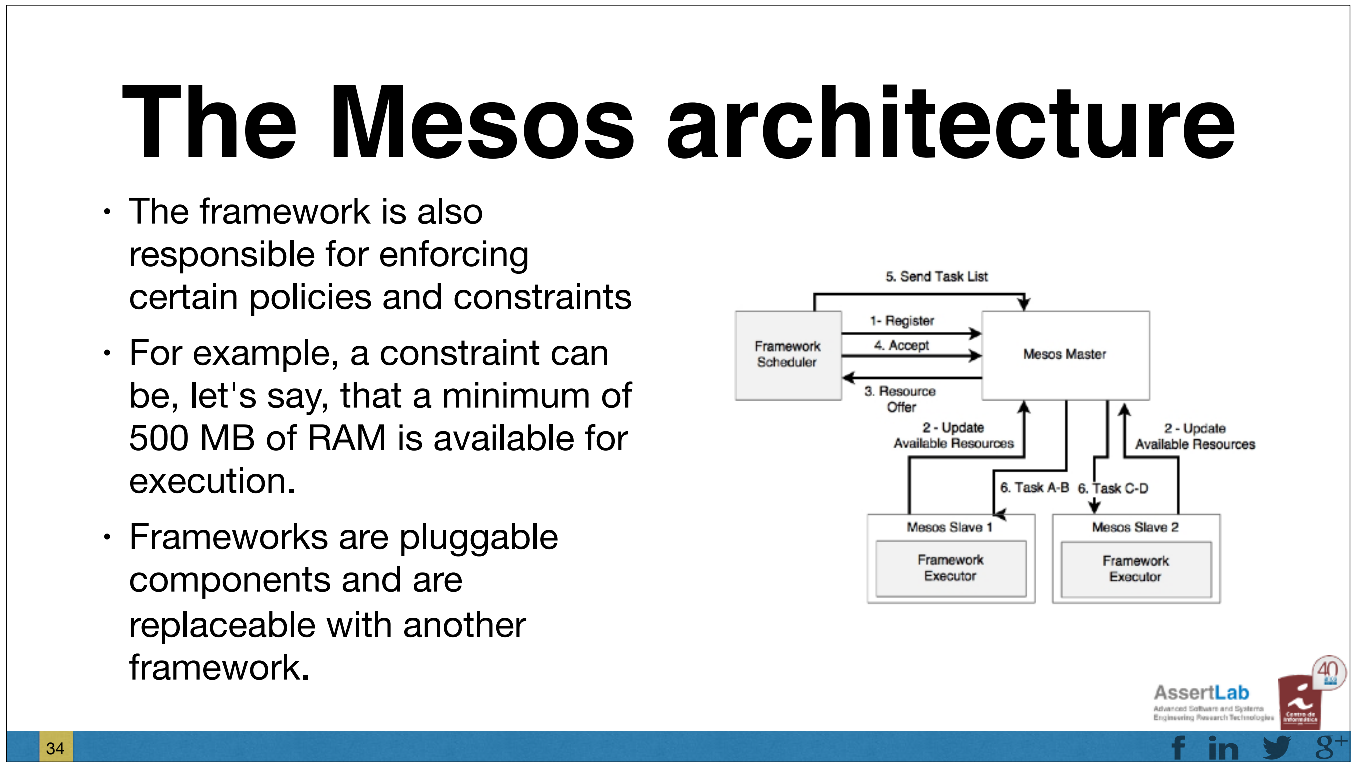


AssertLab
Advanced Software and Systems Engineering Research Technologies

The Mesos architecture

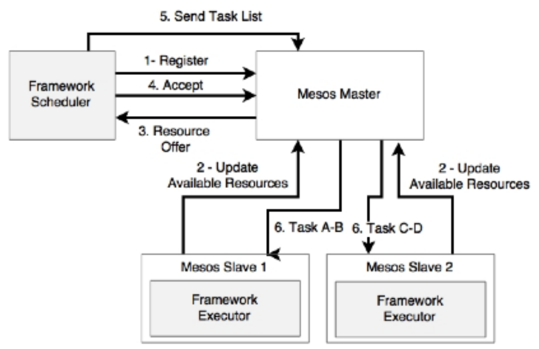
- **Framework**: The Mesos framework is responsible for understanding the application's constraints, accepting resource offers from the master, and finally running tasks on the slave resources offered by the master
  - The Mesos framework consists of two components: the framework scheduler and the framework executor:
    - The scheduler is responsible for registering to Mesos and handling resource offers
    - The executor runs the actual program on Mesos slave nodes





The Mesos architecture

- The framework is also responsible for enforcing certain policies and constraints
- For example, a constraint can be, let's say, that a minimum of 500 MB of RAM is available for execution.
- Frameworks are pluggable components and are replaceable with another framework.



AssertLab

1. The framework registers with the Mesos master and waits for resource offers. The scheduler may have many tasks in its queue to be executed with different resource constraints (tasks A to D, in this example). A task, in this case, is a unit of work that is scheduled—for example, a Spring Boot microservice.
2. The Mesos slave offers the available resources to the Mesos master. For example, the slave advertises the CPU and memory available with the slave machine.
3. The Mesos master then creates a resource offer based on the allocation policies set and offers it to the scheduler component of the framework. Allocation policies determine which framework the resources are to be offered to and how many resources are to be offered. The default policies can be customized by plugging additional allocation policies.
4. The scheduler framework component, based on the constraints, capabilities, and policies, may accept or reject the resource offering. For example, a framework rejects the resource offer if the resources are insufficient as per the constraints and policies set.
5. If the scheduler component accepts the resource offer, it submits the details of one more task to the Mesos master with resource constraints per task. Let's say, in this example, that it is ready to submit tasks A to D.
6. The Mesos master sends this list of tasks to the slave where the resources are available. The framework executor component installed on the slave machines picks up and runs these tasks.

The Mesos architecture

- Mesos supports a number of frameworks, such as:
  - Marathon and Aurora for **long-running** processes, such as web applications
  - Hadoop, Spark, and Storm for **big data** processing
  - Chronos and Jenkins for **batch scheduling**
  - Cassandra and Elasticsearch for **data management**
- We will use Marathon to **run** dockerized microservices

Marathon

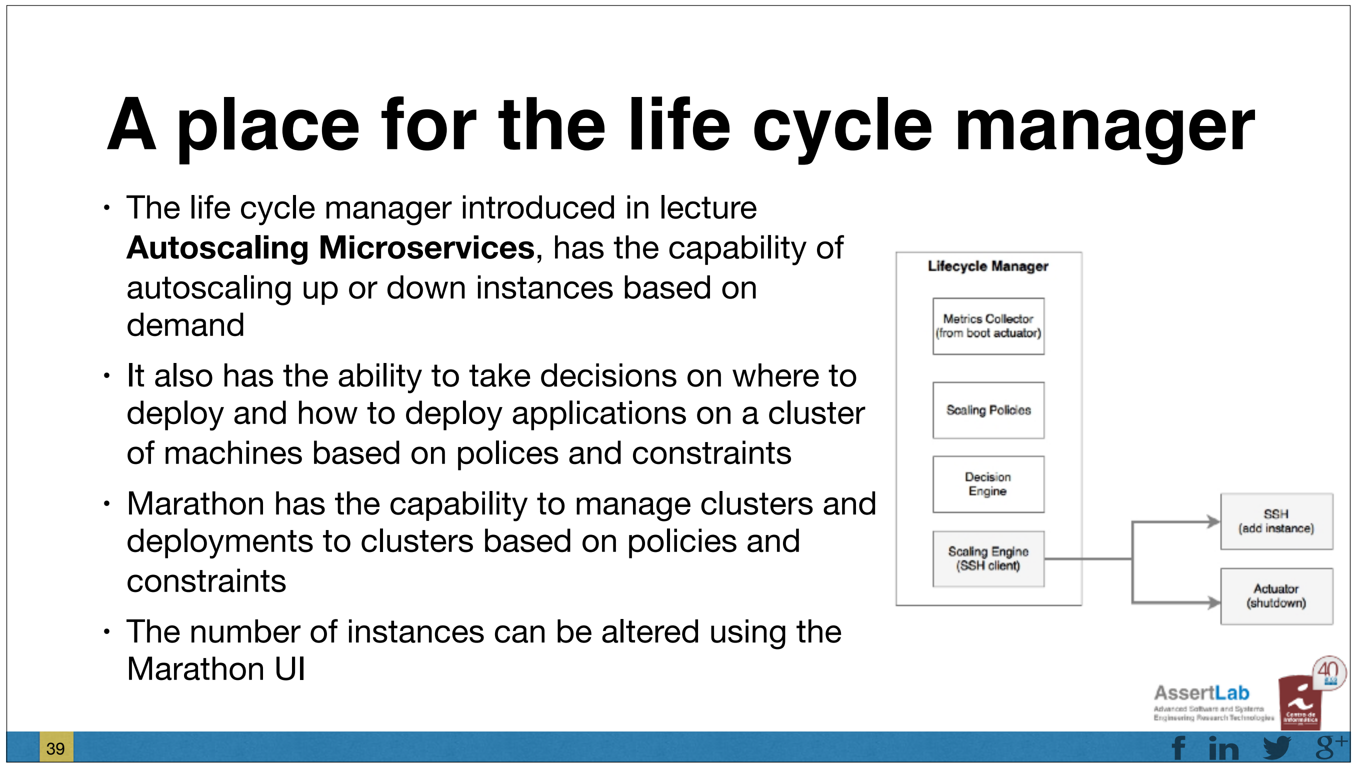
- Marathon is one of the Mesos framework implementations that can run both container as well as noncontainer execution
- Marathon ensures that the service started with Marathon continues to be available even if the Mesos slave it is hosted on fails
- Marathon is written in Scala and is highly scalable
- Offers a UI as well as REST APIs to interact with him, such as the start, stop, scale, and monitoring applications
- Marathon's high availability is achieved by running multiple Marathon instances pointing to a ZooKeeper instance
  - One of the Marathon instances acts as a leader, and others are in standby mode

Marathon

- Some of the basic features of Marathon include:
  - Setting resource constraints
  - Scaling up, scaling down, and the instance management of applications
  - Application version management
  - Starting and killing applications
- Some of the advanced features of Marathon include:
  - Rolling upgrades, rolling restarts, and rollbacks
  - Blue-green deployments

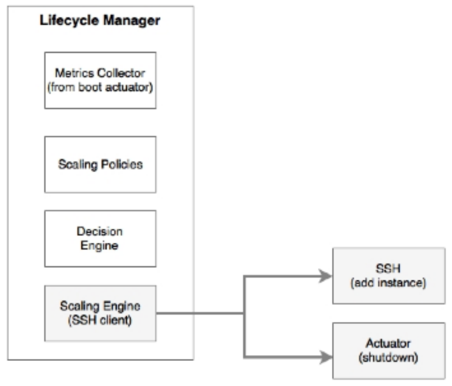
Homework 12

- A guided tour to implementing Mesos and Marathon for BrownField microservices



A place for the life cycle manager

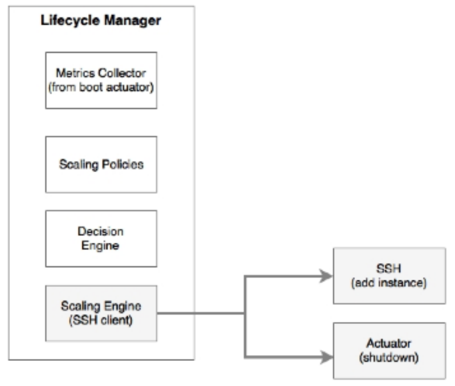
- The life cycle manager introduced in lecture **Autoscaling Microservices**, has the capability of autoscaling up or down instances based on demand
- It also has the ability to take decisions on where to deploy and how to deploy applications on a cluster of machines based on polices and constraints
- Marathon has the capability to manage clusters and deployments to clusters based on policies and constraints
- The number of instances can be altered using the Marathon UI

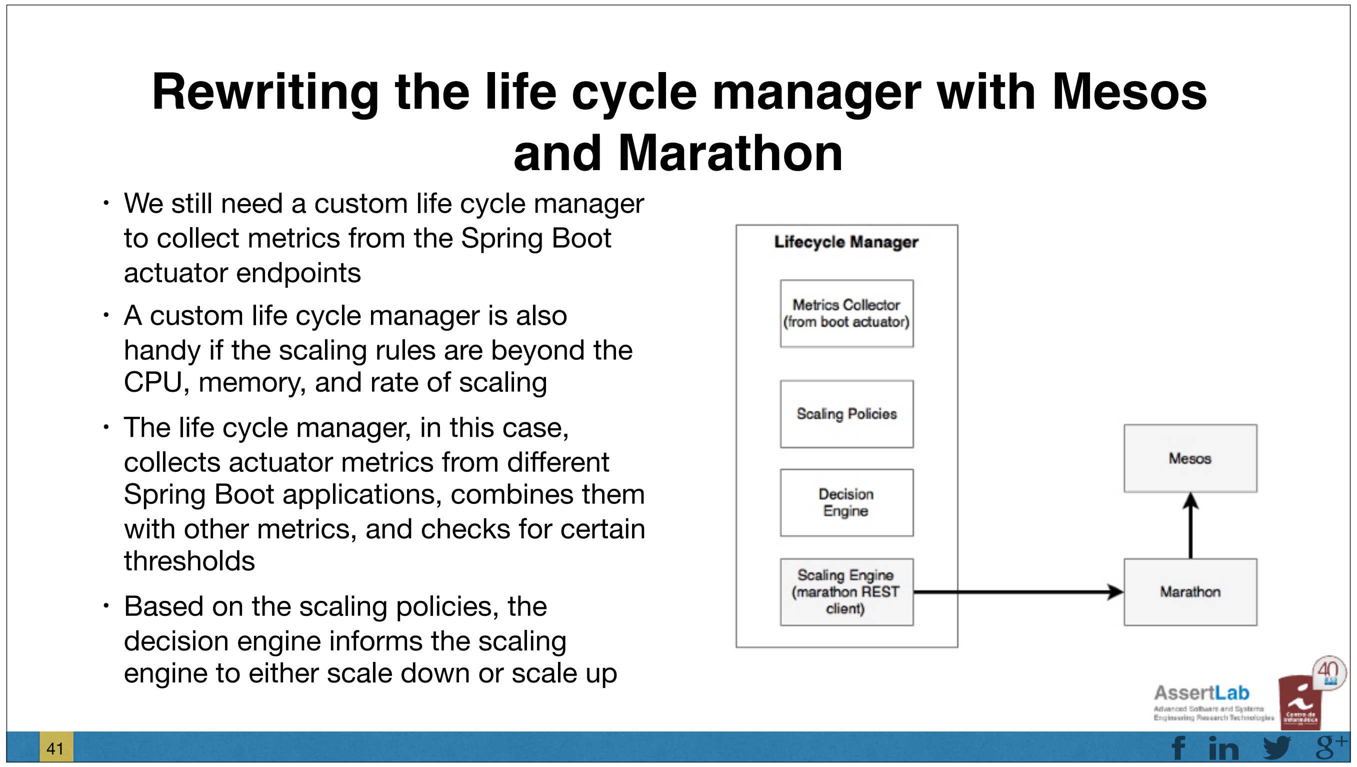


There are redundant capabilities between our life cycle manager and Marathon. With Marathon in place, SSH work or machine-level scripting is no longer required. Moreover, deployment policies and constraints can be delegated to Marathon. The REST APIs exposed by Marathon can be used to initiate scaling functions.

A place for the life cycle manager

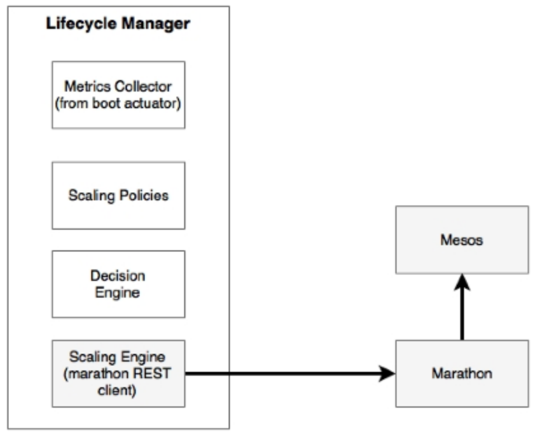
- **Marathon autoscale** is a proof-of-concept project from Mesosphere for autoscaling
- The Marathon autoscale provides basic autoscale features such as the CPU, memory, and rate of request





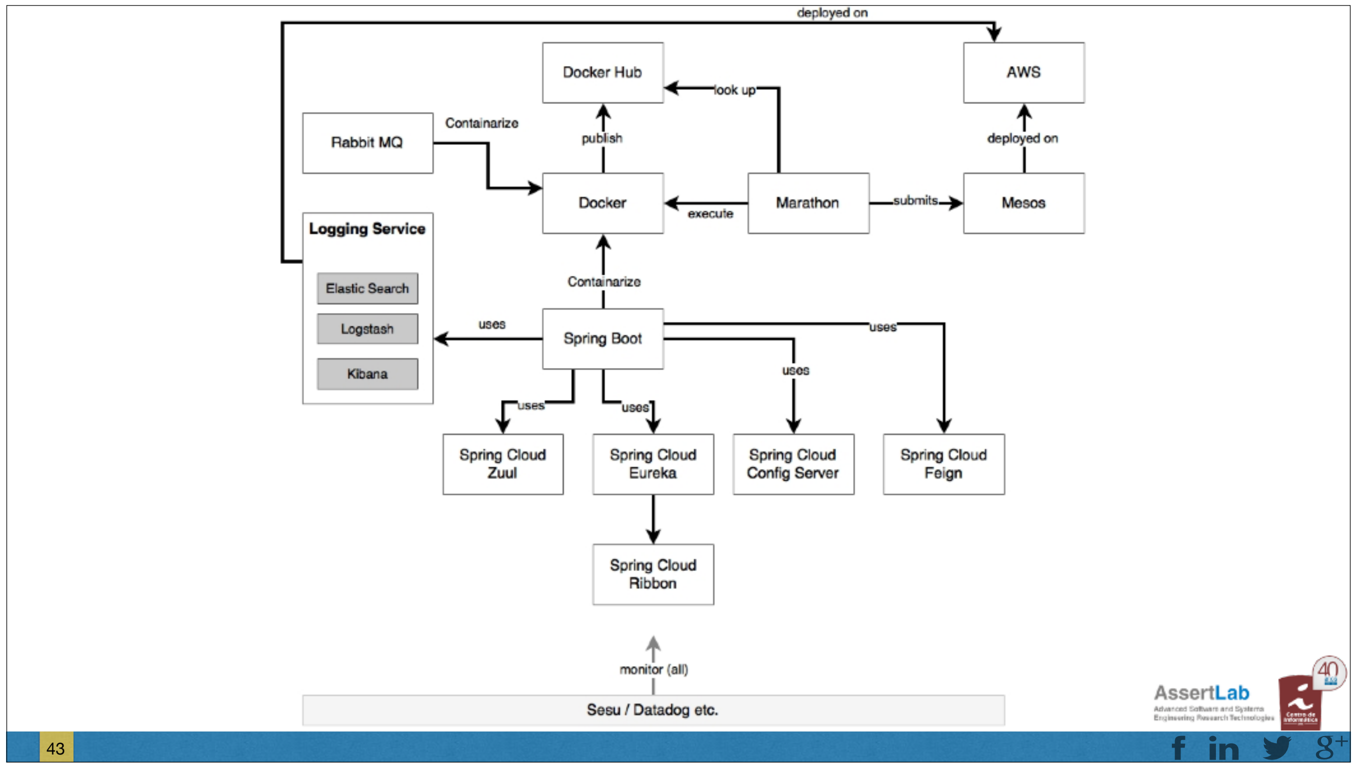
Rewriting the life cycle manager with Mesos and Marathon

- We still need a custom life cycle manager to collect metrics from the Spring Boot actuator endpoints
- A custom life cycle manager is also handy if the scaling rules are beyond the CPU, memory, and rate of scaling
- The life cycle manager, in this case, collects actuator metrics from different Spring Boot applications, combines them with other metrics, and checks for certain thresholds
- Based on the scaling policies, the decision engine informs the scaling engine to either scale down or scale up


Lifecycle Manager
Metrics Collector (from boot actuator)
Scaling Policies
Decision Engine
Scaling Engine (marathon REST client)
Mesos
Marathon

In this case, the scaling engine is nothing but a Marathon REST client. This approach is cleaner and neater than our earlier primitive life cycle manager implementation using SSH and Unix scripts

The technology metamodel



We have covered a lot of ground on microservices with the BrownField PSS microservices. The following diagram sums it up by bringing together all the technologies used into a technology metamodel