Big Data Engineer Bootcamp

Part 2

Agenda

- Introduction to Docker
- Introduction to Redis
- Introduction to Node.js
- Introduction to Spark
- Introduction to Mesos
- Q&A

Introduction to Docker

Build, Ship, and Run Any App, Anywhere

Agenda

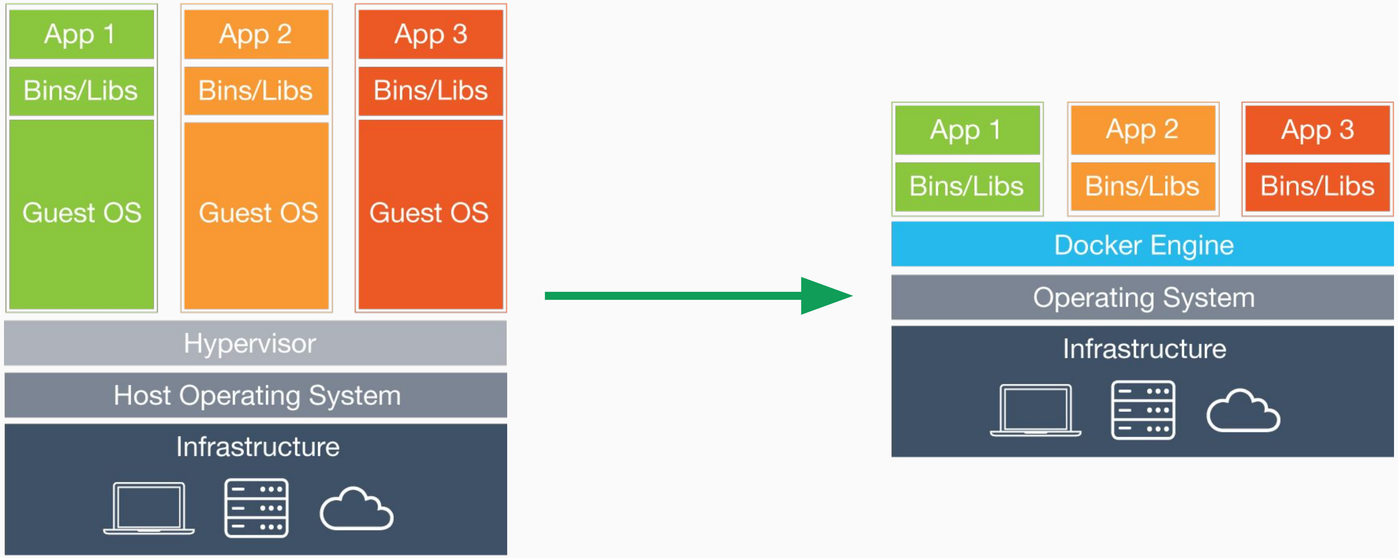
- **Use Cases**
- What is Docker
- Architecture
- Docker Usage

Consistent Deployment Model

- In the industry, a software will go through multiple stages
  - Development machine
  - QA - testing
  - Staging - integrate with other teams
  - Production - handles real traffic

	Development Machine	QA Environment	Staging Environment	Production Environment
data-producer.py	?	?	?	?

Increase Resource Utilization



Docker Benefits

- Faster developer onboarding
- No vendor lock-in
- Eliminate environment inconsistencies
- Ship applications faster
- Scale quickly
- Easily remediate issues

Agenda

- Use Cases
- **What is Docker**
- Architecture
- Docker Usage

What is Docker?

- A tool to package and deploy applications inside containers
  - Containers are isolated environments
- Developed by Solomon Hykes in Dotcloud
- Open-sourced in March 2013, written in Go
- Grown into a platform
  - Docker Compose, Docker Swarm
  - Docker Image Hosting
  - Container Hosting

Agenda

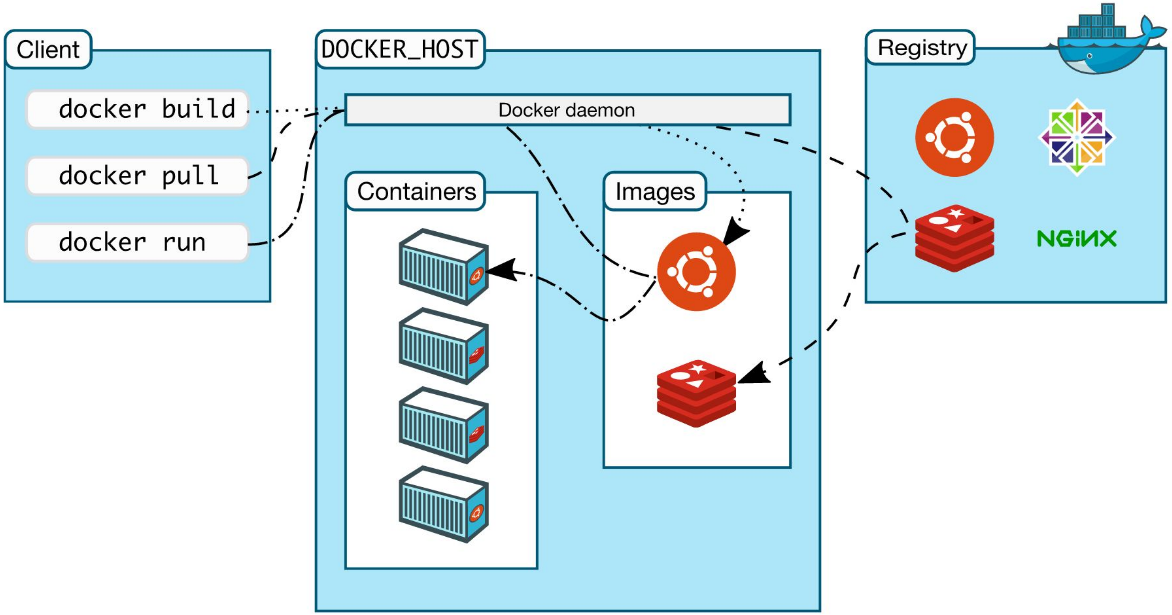
- Use Cases
- What is Docker
- **Architecture**
- Docker Usage
- Hands on

Architecture

- **Concepts**
- Internal

Client Server Architecture

- Docker Client
- Docker Daemon
- Docker Registry



Docker Client

- A small client to communicate with Docker Daemon
    - User interact with Docker Client to perform tasks
- Can connect to any remote Docker Daemon

Docker Daemon

- A background daemon running on host servers
  - a small server takes requests from Docker Client and dispatch/route to corresponding handler
  - an engine takes the requests and manipulates containers such as creating container, etc

Docker Registry

- A warehouse for container images
  - Similar to github
- Talks to Docker Daemon to handle image related requests from Docker Client
- You can use public Docker Registry (which is Dockerhub) or private Docker Registry

Architecture

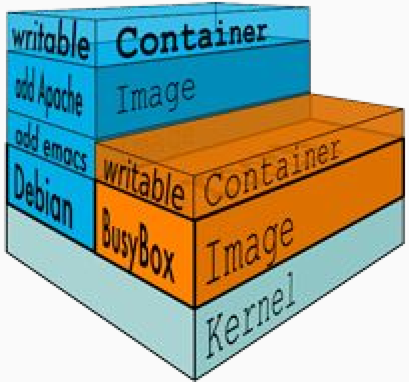
- Concepts
- **Internal**

Docker Image

- The basis of containers
- Contains a number of layers
- Use union filesystem to combine layers to form a single read-only file system
- Docker Images are built from
  - run a base container, run commands within the container and commit, push the changes like git
  - base images (such as ubuntu) and set of instructions
- Instructions are stored in a file called Dockerfile

Docker Image

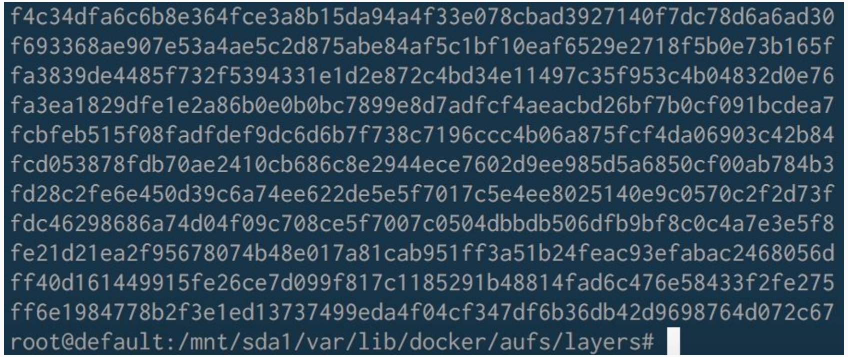
- In Docker, each layer is described by:
  - Meta data of the layer, in JSON format
  - Image Filesystem changeset
  - Image ID, such as: a9561eb1b190625c9adb5a9513e72c4dedafc1cb2d4c5236c9a6957ec7dfd5a9



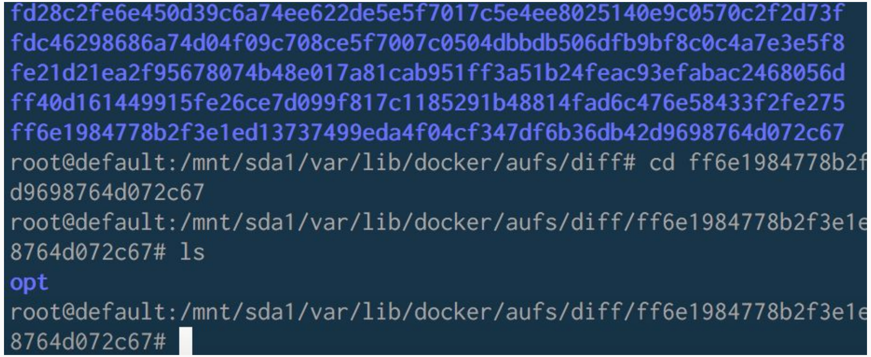
Docker Image

- Look inside /var/lib/docker/aufs/layers

- Look inside /var/lib/docker/aufs/diff

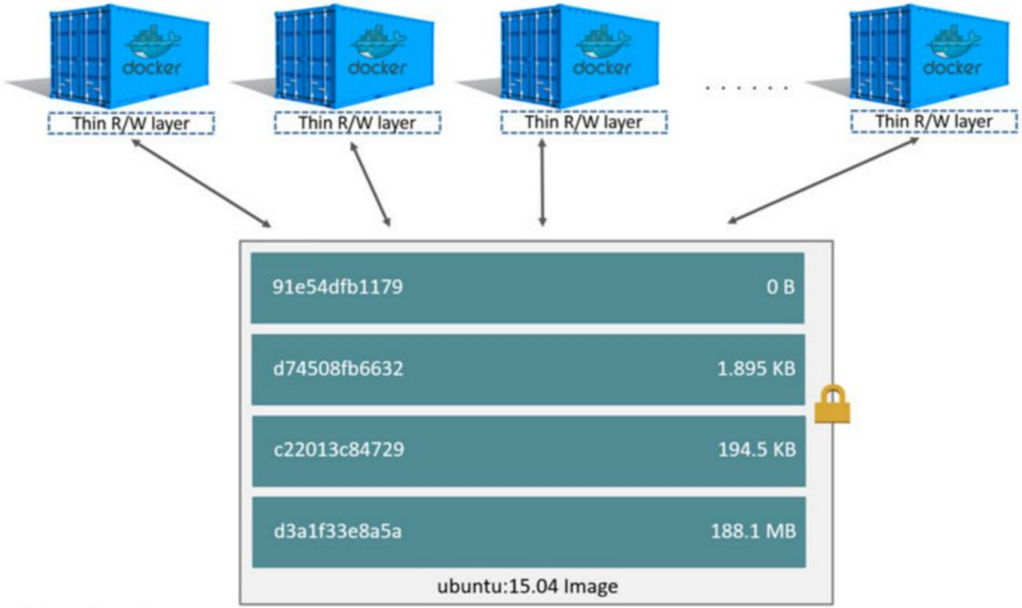


```
f4c34dfa6c6b8e364fce3a8b15da94a4f33e078cbad3927140f7dc78d6a6ad30
f693368ae907e53a4ae5c2d875abe84af5c1bf10eaf6529e2718f5b0e73b165f
fa3839de4485f732f5394331e1d2e872c4bd34e11497c35f953c4b04832d0e76
fa3ea1829dfe1e2a86b0e0b0bc7899e8d7adfcf4aeacbd26bf7b0cf091bcdea7
fcbfeb515f08fadfdef9dc6d6b7f738c7196ccc4b06a875fcf4da06903c42b84
fcd053878fdb70ae2410cb686c8e2944ece7602d9ee985d5a6850cf00ab784b3
fd28c2fe6e450d39c6a74ee622de5e5f7017c5e4ee8025140e9c0570c2f2d73f
fdc46298686a74d04f09c708ce5f7007c0504dbbdb506dfb9bf8c0c4a7e3e5f8
fe21d21ea2f95678074b48e017a81cab951ff3a51b24feac93efabac2468056d
ff40d161449915fe26ce7d099f817c1185291b48814fad6c476e58433f2fe275
ff6e1984778b2f3e1ed13737499eda4f04cf347df6b36db42d9698764d072c67
root@default:/mnt/sda1/var/lib/docker/aufs/layers#
```



```
fd28c2fe6e450d39c6a74ee622de5e5f7017c5e4ee8025140e9c0570c2f2d73f
fdc46298686a74d04f09c708ce5f7007c0504dbbdb506dfb9bf8c0c4a7e3e5f8
fe21d21ea2f95678074b48e017a81cab951ff3a51b24feac93efabac2468056d
ff40d161449915fe26ce7d099f817c1185291b48814fad6c476e58433f2fe275
ff6e1984778b2f3e1ed13737499eda4f04cf347df6b36db42d9698764d072c67
root@default:/mnt/sda1/var/lib/docker/aufs/diff# cd ff6e1984778b2f
d9698764d072c67
root@default:/mnt/sda1/var/lib/docker/aufs/diff/ff6e1984778b2f3e1e
8764d072c67# ls
opt
root@default:/mnt/sda1/var/lib/docker/aufs/diff/ff6e1984778b2f3e1e
8764d072c67#
```

Docker Container



Thin R/W layer

Thin R/W layer

Thin R/W layer

Thin R/W layer

91e54dfb1179 0 B

d74508fb6632 1.895 KB

c22013c84729 194.5 KB

d3a1f33e8a5a 188.1 MB

ubuntu:15.04 Image

graphdb

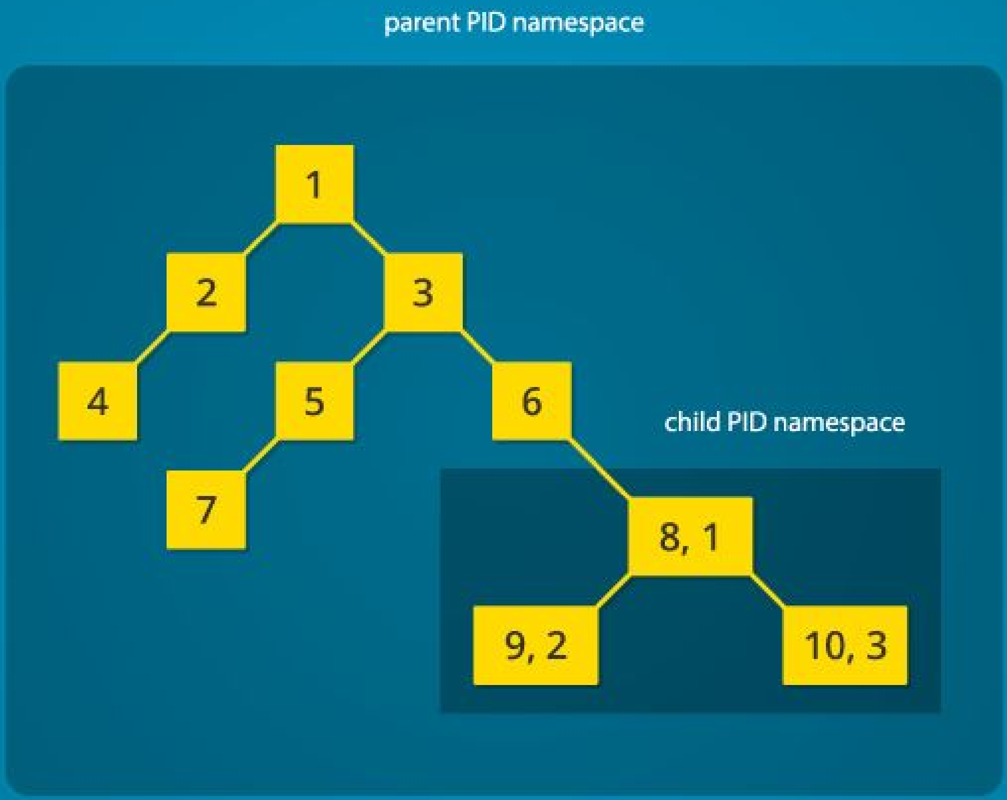
- Manage local Docker Image and their relationships
- Internally use SQLite

Dockerfile Example

```
FROM mhart/alpine-node:base
ADD . .
EXPOSE 3000
CMD ["node", "index.js"]
```

namespace

- A linux kernel functionality that can
    - perform resource limiting
    - resource prioritization
    - resource accounting
- Docker uses cgroup to achieve resource limiting



cgroup

- A linux kernel functionality that can
  - perform resource limiting
  - resource prioritization
  - resource accounting

- Docker uses cgroup to achieve resource limiting

Docker Driver

- Customize the execution environment of Docker Containers
- graph driver - related to storage
  - aufs, devicemapper, btrfs, zfs, and overlay
- network driver - related to network
  - bridge, ip, port
- exec driver - related to the execution of container
  - LXC, runC

runC

- An abstraction layer between Docker Driver and Linux kernel
- It interfaces with Linux kernel functionalities such as:
  - namespaces
  - cgroups
  - capabilities
  - file system access controls

Agenda

- Use Cases
- What is Docker
- Architecture
- **Docker Usage**

Docker Usage

- Speed Up Workflow
- Handle Spike Traffics

Speed Up Workflow

- Push from dev machine to production in seconds
- Deliver functionalities to end user multiple times a day
  - CI/CD + Docker

	Development Machine	QA Environment	Staging Environment	Production Environment
data-producer.py	!	!	!	!

Handle Spike Traffics

- Docker is perfect for stateless tasks/jobs
- Starts up within second

Further Reading

- Docker Cheetsheet: https://github.com/wsargent/docker-cheat-sheet
- namespace: https://en.wikipedia.org/wiki/Linux_namespaces
- cgroup: https://en.wikipedia.org/wiki/Cgroups

Introduction to Redis

Swiss Knife Data Structure

What is Redis

- Open source in-memory data structure store
    - Database
    - Cache
    - Message Queue

- Developed by Salvatore Sanfilippo
- Implemented in C, high performance
- Super clean API + data structure

Supported Data Structure

- Strings
- Lists
- Sets
- Sorted Sets
- Hashes (think of it as map)
- Bitmap
- Hyperloglog

Supported API

- Create/Read/Update/Delete
  - SET key value
  - GET key
  - LPOP key
- TTL
  - SETEX key value ttl
- And many more (http://redis.io/commands)

Use Cases

- Use Redis as LRU cache
- Use Redis as non-critical message queue

- Basically think of Redis as your Leetcode as a server ;)

Further Reading

- Redis official documentation: http://redis.io/documentation
- Little Redis Book: http://openmymind.net/redis.pdf
- Salvatore Sanfilippo’s blog: http://antirez.com/latest/0 ****

Introduction to Node.js

Swiss Knife Data Structure

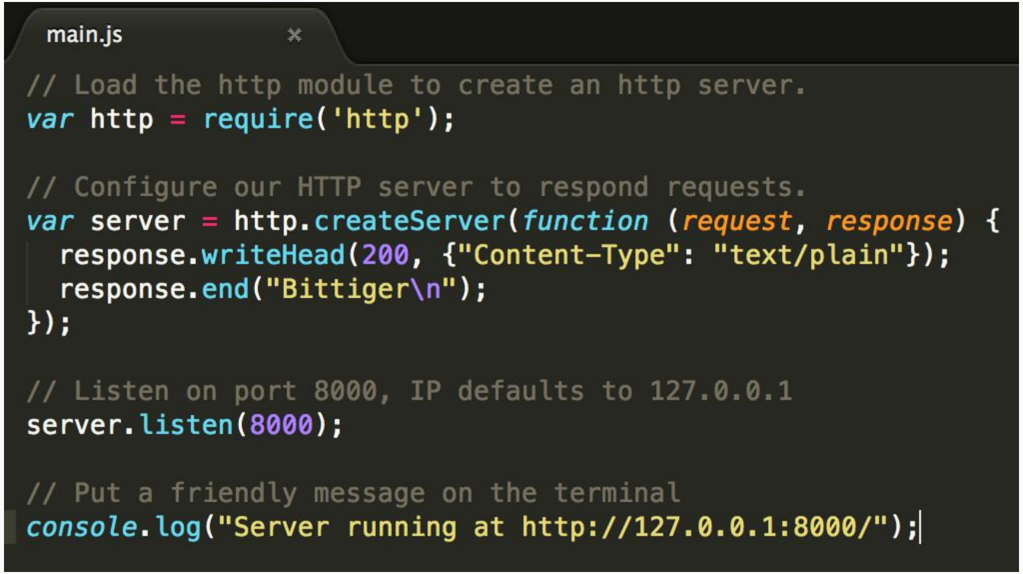
What is Node.js

- Open source platform for service-side web applications
  - Event-driven architecture
  - Async IO

- Developed by Ryan Dahl
- Instant popularity for stack cohesion
  - Emerge of full-stack engineer

Sample Node.js Web Application

- Functions are first level citizen in javascript
- Callback functions are heavily used in Node.js



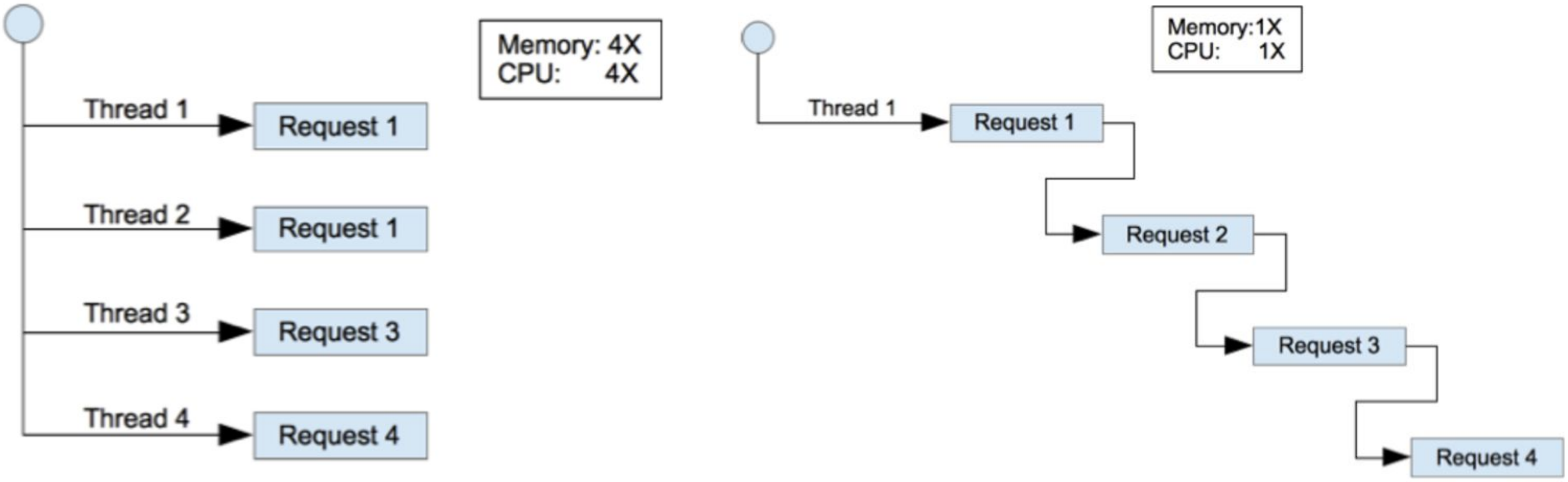
```
// Load the http module to create an http server.
var http = require('http');

// Configure our HTTP server to respond requests.
var server = http.createServer(function (request, response) {
  response.writeHead(200, {"Content-Type": "text/plain"});
  response.end("Bittiger\n");
});

// Listen on port 8000, IP defaults to 127.0.0.1
server.listen(8000);

// Put a friendly message on the terminal
console.log("Server running at http://127.0.0.1:8000/");
```

Comparison between J2EE and Node.js

- In web applications, majority of the work is IO

	Concurrency Model	I/O Model
J2EE	Multiple thread	Synchronous IO
Node.js	Single thread	Asynchronous IO

Multi-thread vs Single-thread



Memory: 4X
CPU: 4X

Thread 1 → Request 1

Thread 2 → Request 1

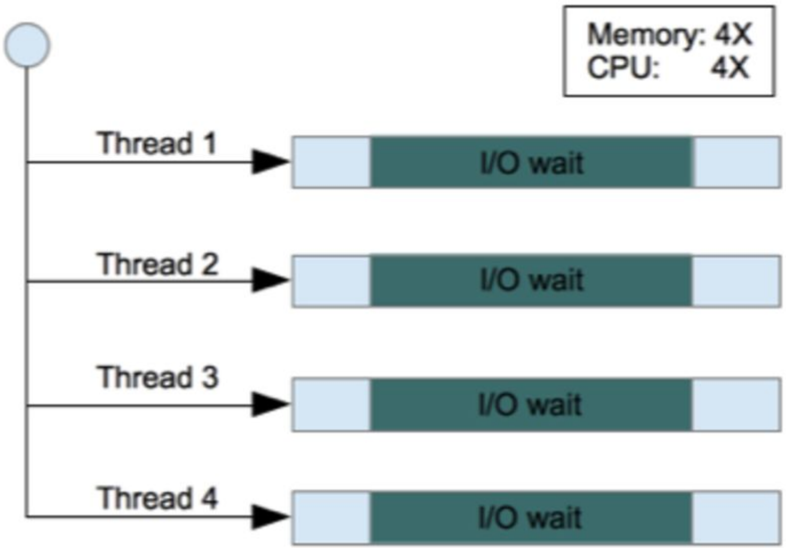
Thread 3 → Request 3

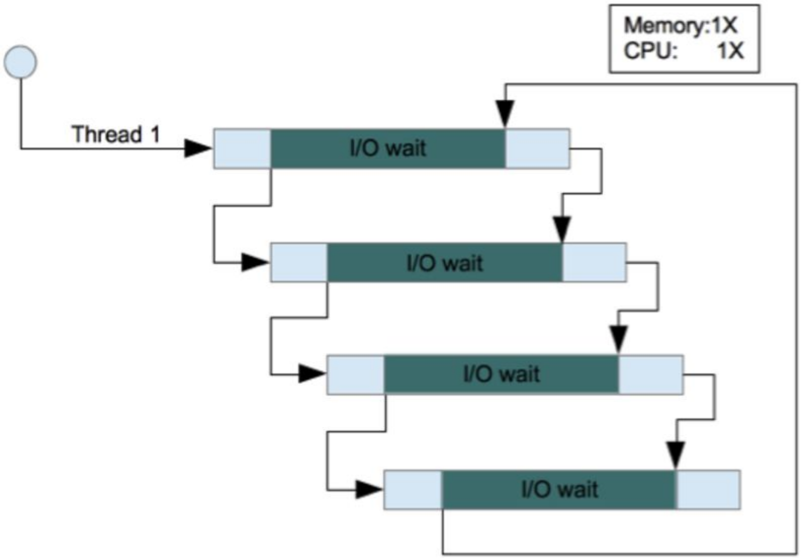
Thread 4 → Request 4

Memory:1X
CPU: 1X

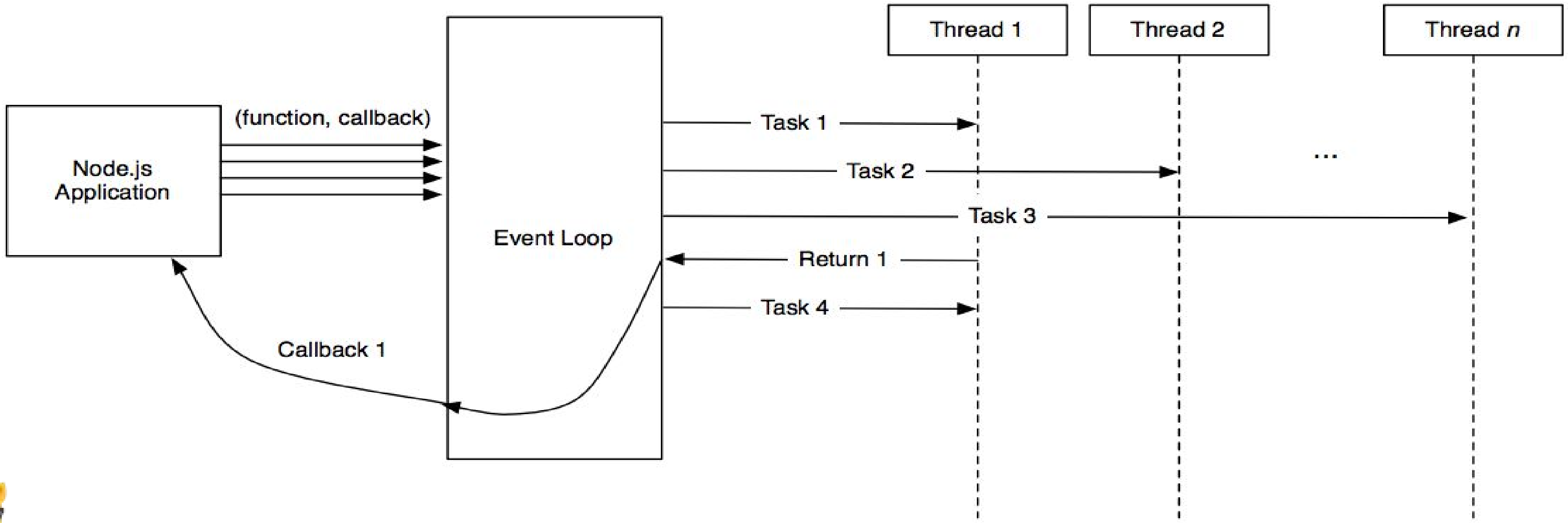
Thread 1 → Request 1 → Request 2 → Request 3 → Request 4

Synchronos IO vs Asynchronos IO





Eventloop


Thread 1
Thread 2
Thread *n*
Node.js Application
(function, callback)
Event Loop
Task 1
Task 2
...
Task 3
Return 1
Task 4
Callback 1

Further Reading

- Node.js Documentation: https://nodejs.org/api/
- Node.js Beginners Guide: http://nodeguide.com/beginner.html
- Deep dive into Node.js Architecture: http://www.journaldev.com/7462/node-js-architecture-single-threaded-event-loop

Introduction to Spark

Lightning-fast Cluster Computing

Agenda

- **Use Cases**
- What is Spark
- Architecture
- Spark Usage
- Hands on

Example Compute Problem

- Give a collection of fruits
- Count the quantity of each type of fruit

Example Compute Problem

- Naive Approach
  - For every type of fruit, find all of the items
  - Count the number of items
- Divide the Conquer
  - Split the fruits into small chunks
  - Count on the small chunks
  - Aggregate the results

Distributed Computing

- Divide and Conquer fits perfectly with Distributed Model
- Hard to implement
    - Network issue
    - Slave unavailable
    - No clean set of APIs

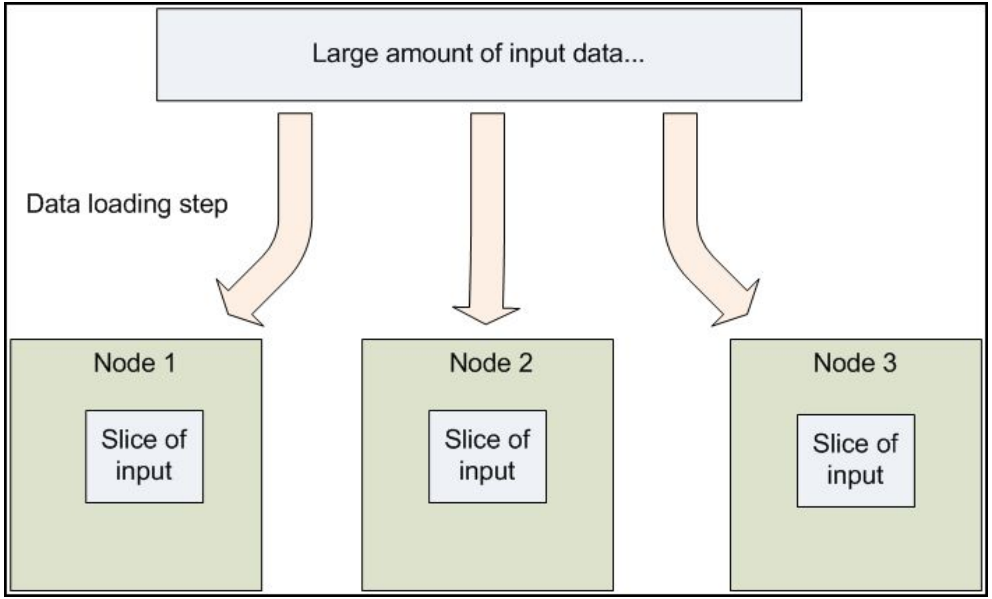
Apache Hadoop

- Open source distributed computing framework
  - Simply programming model
  - High resiliency to hardware failure/network failure

- Developed by Doug Cutting at Yahoo!
- Based on Google GFS and MapReduce paper
- Ground-breaking project from Apache Software Foundation

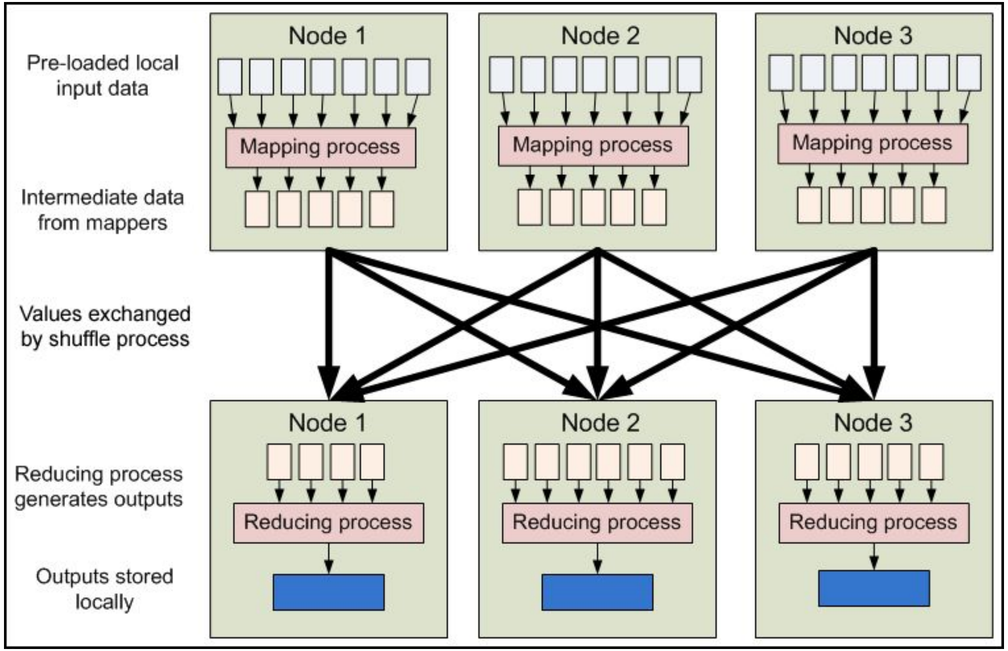
Apache Hadoop HDFS

- Handles data storage aspect of big data
- Master-slave architecture
- Data split into different slaves



Apache Hadoop MapReduce

- Slave nodes run computation on the sub-data set -> **MAP**
- Intermediate data are saved temporarily and then sorted
- Slave nodes run aggregation of sub-data set result -> **REDUCE**



Problem with Apache Hadoop

- Massive disk IO
    - Mapper write intermediate data into disk
    - Transferred through network for shuffling
    - Reducer load intermediate data from disk

Agenda

- Use Cases
- **What is Spark**
- Architecture
- Spark Usage
- Hands on

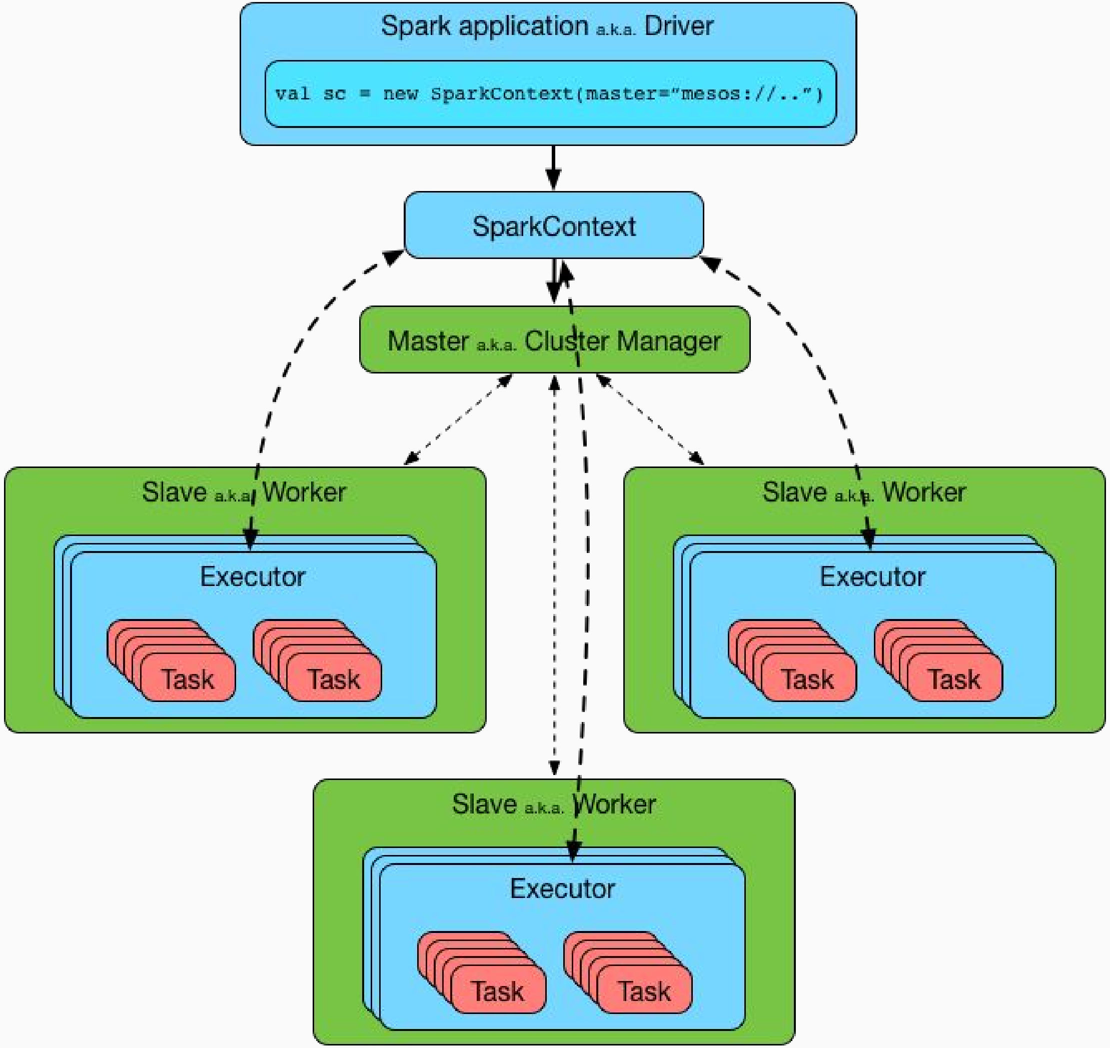
What is Spark

- Open source cluster computing framework
  - Respond to limitations of Apache Hadoop
  - Computation optimization
  - In memory computing
- Developed at UC Berkeley by Matei Zaharia

Agenda

- Use Cases
- What is Spark
- **Architecture**
- Spark Usage
- Hands on

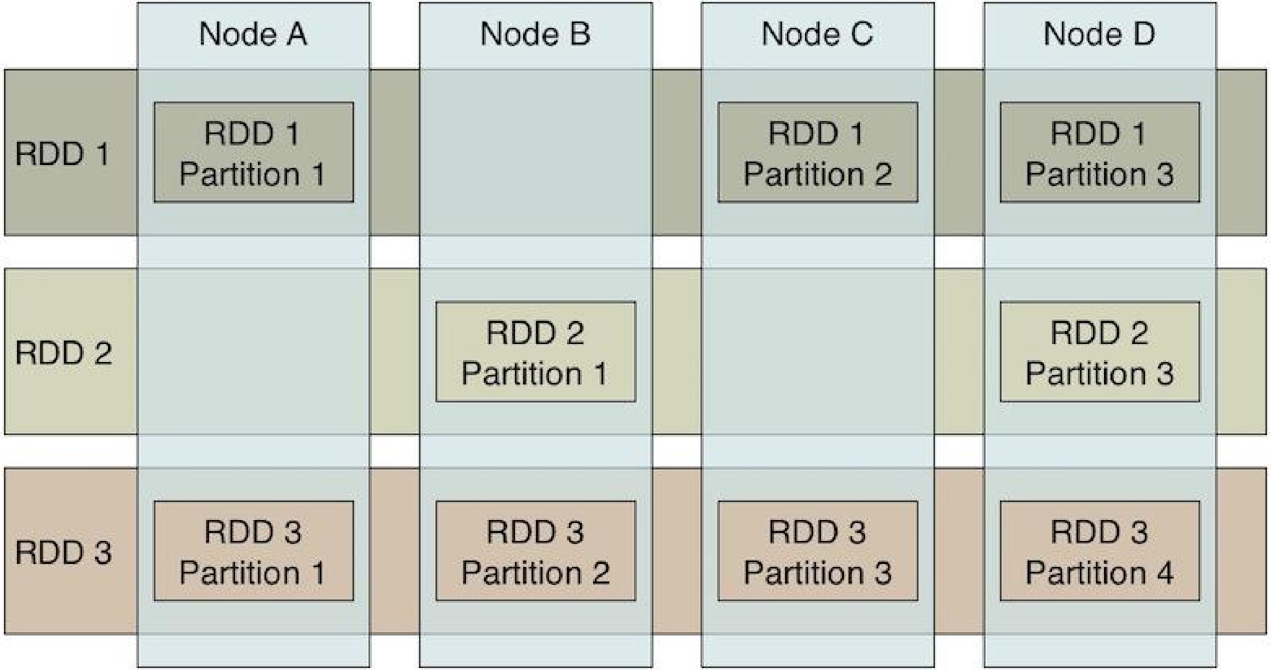
Master Slave Mode



Resilient Distributed Datasets - RDD

- How Spark represents data
- RDD for one data set spread across the Spark cluster

Resilient Distributed Datasets - RDD



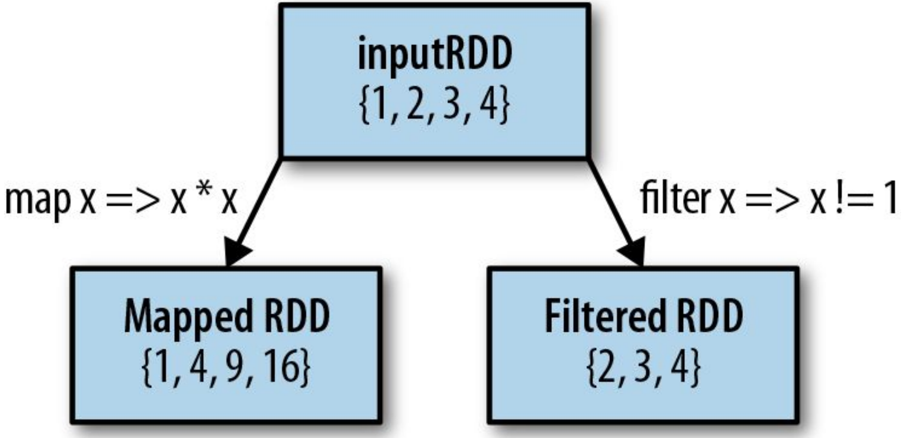
	Node A	Node B	Node C	Node D
RDD 1	RDD 1 Partition 1		RDD 1 Partition 2	RDD 1 Partition 3
RDD 2		RDD 2 Partition 1		RDD 2 Partition 3
RDD 3	RDD 3 Partition 1	RDD 3 Partition 2	RDD 3 Partition 3	RDD 3 Partition 4

Resilient Distributed Datasets - RDD

- RDDs are immutable and readonly
- Can only be built by
  - Load data from raw storage
  - Transform from other RDD

RDD Transformation

- map
- filter
- flatMap
- mapPartitions
- sample
- union
- intersection
- ….



RDD Representation

- Each RDD has the following information
  - A set of partitions
  - A set of dependencies on parent RDD
  - A function for computing from its parents
  - Metadata about data placement

Lazy Evaluation

- Transformation won’t actually perform calculation
- Aggregate compute steps for optimization
- Actual computation happens at action step

RDD Action

- collect
- count
- countByValue
- reduce
- top
- ...

RDD Persistence

- You can instruct Spark to cache certain RDD
- Configurable
  - Memory
  - Disk
  - Mix of both

Wordcount Example

- `text = sc.textFile(‘shakespeare.txt’)`
- `counts = text.flatMap(lambda line: line.split(“ “)).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b)`
- `counts.collect()`

Agenda

- Use Cases
- What is Spark
- Architecture
- **Spark Usage**
- Hands on

Spark Usage

- Machine Learning Algorithms
- Stream Processing

Machine Learning Algorithms

- Many Machine Learning Algorithms runs iteratively on data sets
- Perfect to run on Spark
  - Persist/Cache RDD
  - DAG computation step optimization

Stream Processing

- The Spark Stream library
  - Provides API to run computation over data stream
  - Integration with common data sources
    - Kafka
    - Flume
    - HDFS
    - S3

Introduction to Mesos

Turn Your Data Center Into One Giant Computer

Agenda

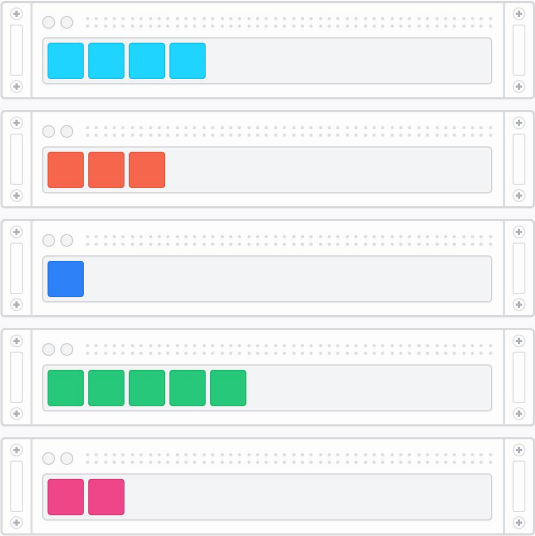
- **Use Cases**
- What is Mesos
- Architecture
- Mesos Usage

System Are Getting Complicated

- More and tools/frameworks
  - Nodejs, J2EE, Ruby on Rails, Django, Flask, etc
  - MySQL, HBase, Cassandra, etc
  - Zookeeper, Etcd, etc
  - Hadoop, Spark, Samza, etc
- Mix of workloads
  - Long running services
  - Batch processes

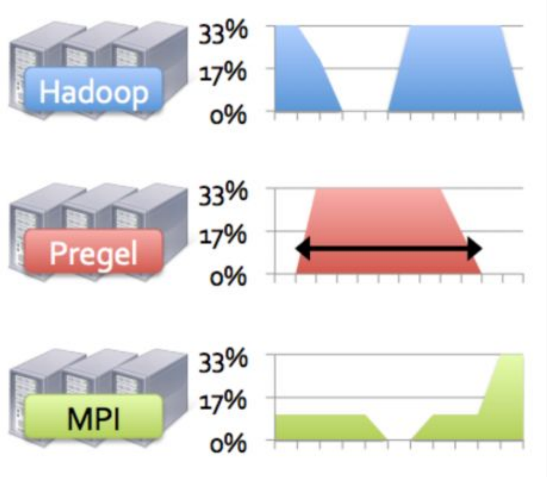
Current Deployment Model

- Static cluster/data center partition
  - group similar apps into one server
- High operation code
- Vendor lock-in

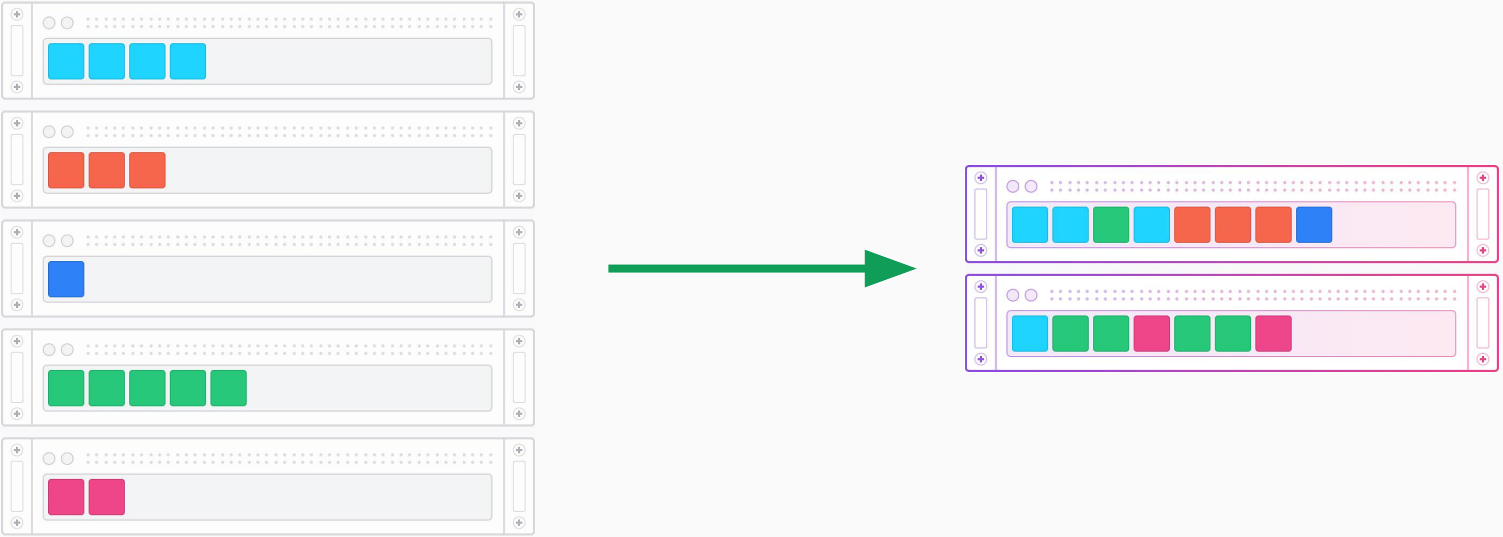


Low Resource Utilization

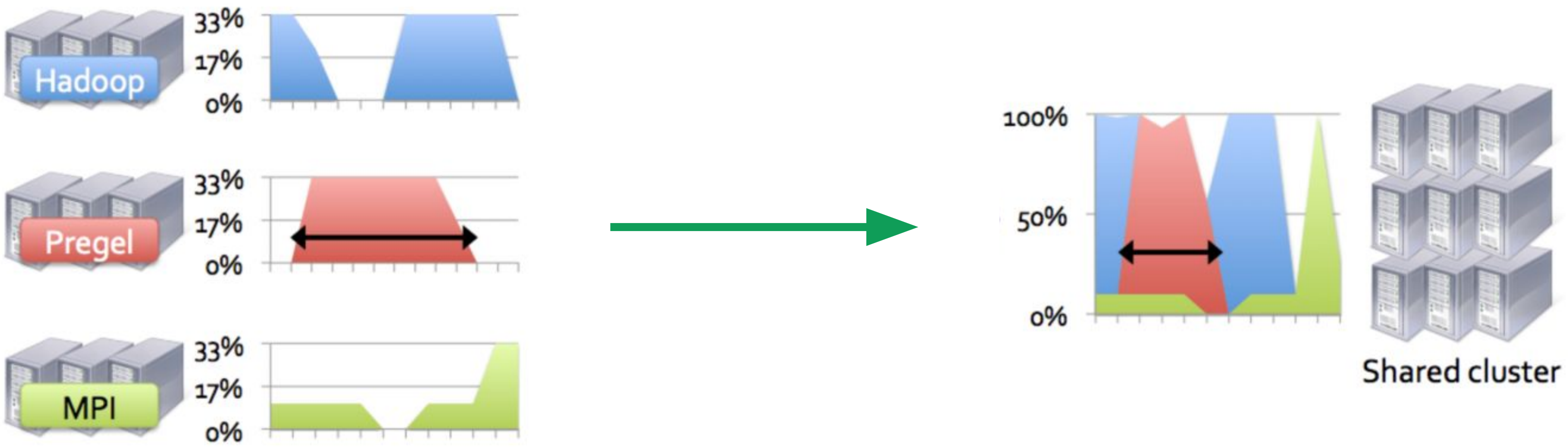
- Different tasks generate different loads on machine
- Different tasks have different resource requirements
  - Memory bound - in memory data processing
  - CPU bound - rendering
  - IO bound - network apps



Management Layer for Data Center



Management Layer for Data Center



Agenda

- Use Cases
- **What is Mesos**
- Architecture
- Mesos Usage

What is Mesos

- An open-source cluster manager that
    - Turn datacenter/cluster into one computer
    - Provide simple API
    - Hide internal complex infrastructure from applications
- By Benjemin Hinderman, Andy Konwinski, and Matei Zaharia in UC Berkeley
- Inspired by Google Borg, matured in Twitter
- Apache Foundation top level project

Agenda

- Use Cases
- What is Mesos
- **Architecture**
- Mesos Usage

Architecture

- **Design Strategy**
- Concepts
- Internal

Two-level Scheduling

- Goal of Mesos is to schedule everything
  - Spark jobs
  - Jenkins build jobs
- Frameworks are different
  - API
  - Lifecycle
  - Scheduling requirements
- Best thing to do is to do nothing
  - Highly scalable
  - Small codebase
  - Easy to customize

Fairness

- How to fairly match jobs/tasks with resources
- How to work with different resource types
    - Memory
    - CPU

Architecture

- Design Strategy
- **Concepts**
- Internal

Framework

- Framework in Mesos is a distributed application developed with Mesos API
- A framework handles a type of workload or job
  - Hadoop
  - Spark
  - Jenkins
  - Chronos
  - Marathon

Resources and Attributes

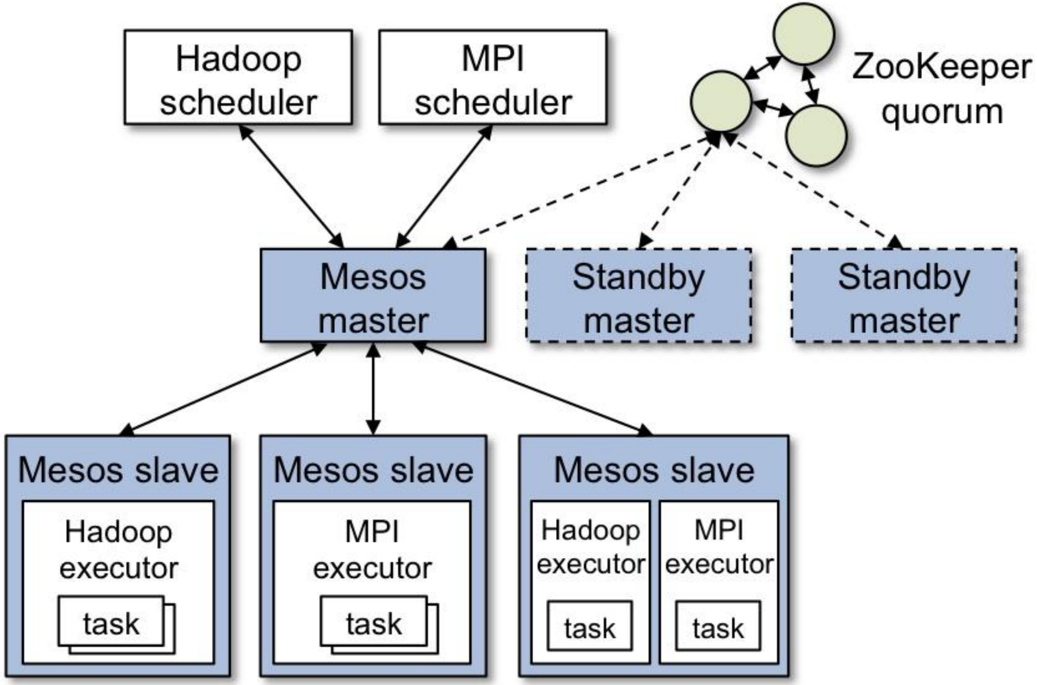
- Resources represent what a slave has to offer
  - cpu
  - mem
  - disk
  - ports
  - --resources=’cpus:24;mem:24576;disk:409600;ports:[21000-24000];types:{a,b,c}’
- Attributes are kv pairs that Mesos passes along when sending offer to framework
  - --attributes=’rack:abc;zone:west;os:centos5;level:10;keys:[1000-15000]’

Architecture

- Design Strategy
- Concepts
- **Internal**

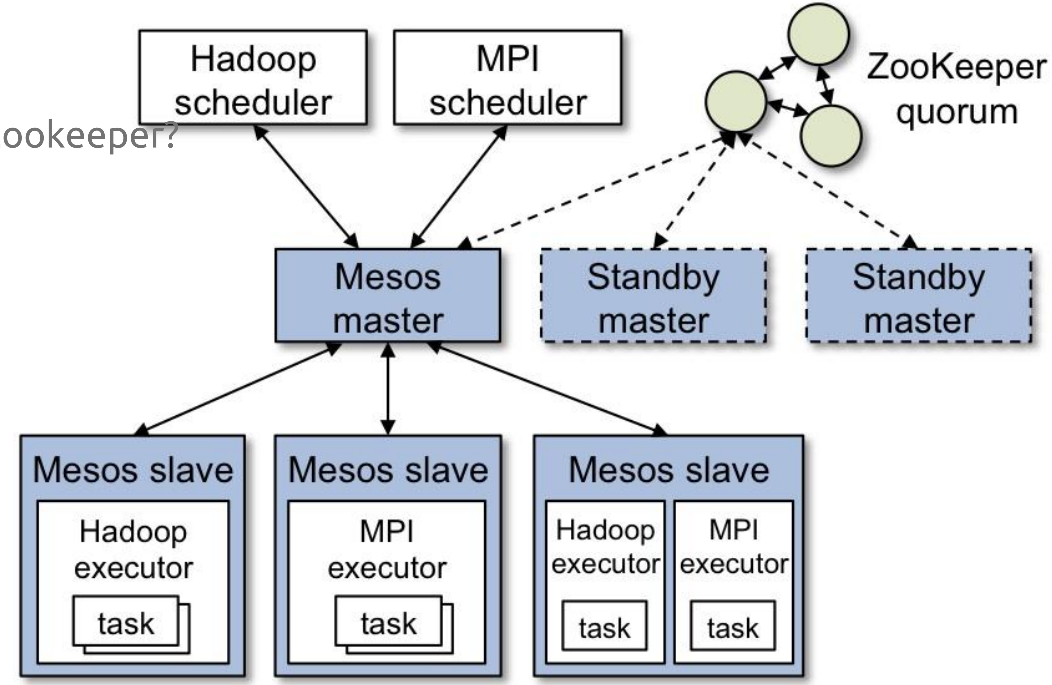
Main Components

- Mesos Master
- Mesos Slave
- Zookeeper
- Frameworks
  - Scheduler + Executor
- Protocol Buffer + libprocess



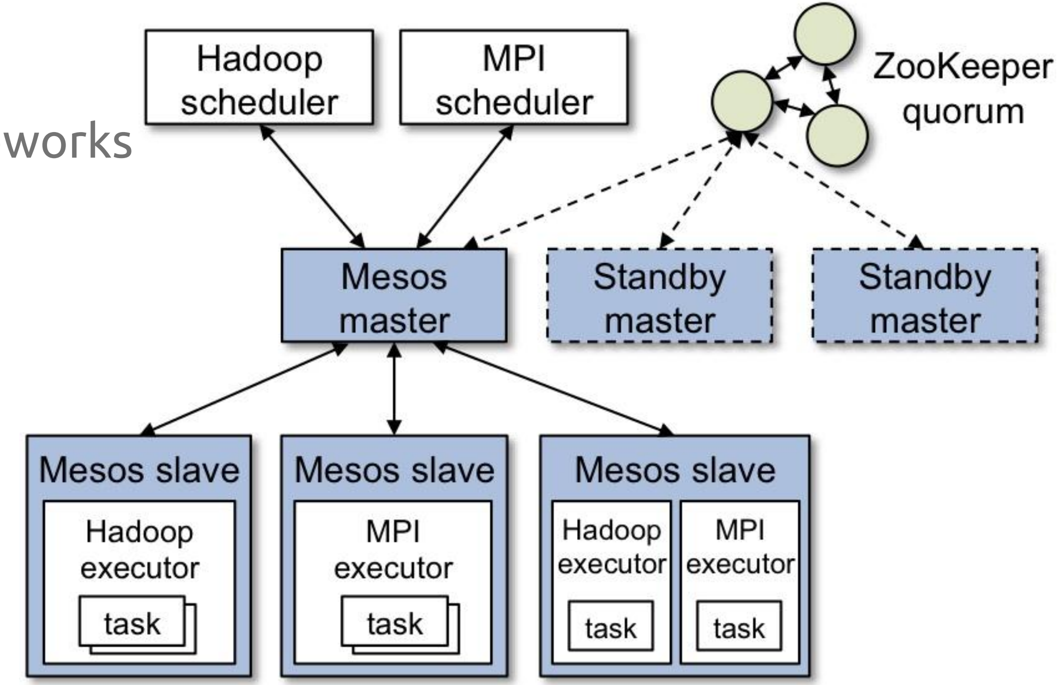
Zookeeper

- Handles leader election
  - How to do leader election with Zookeeper?



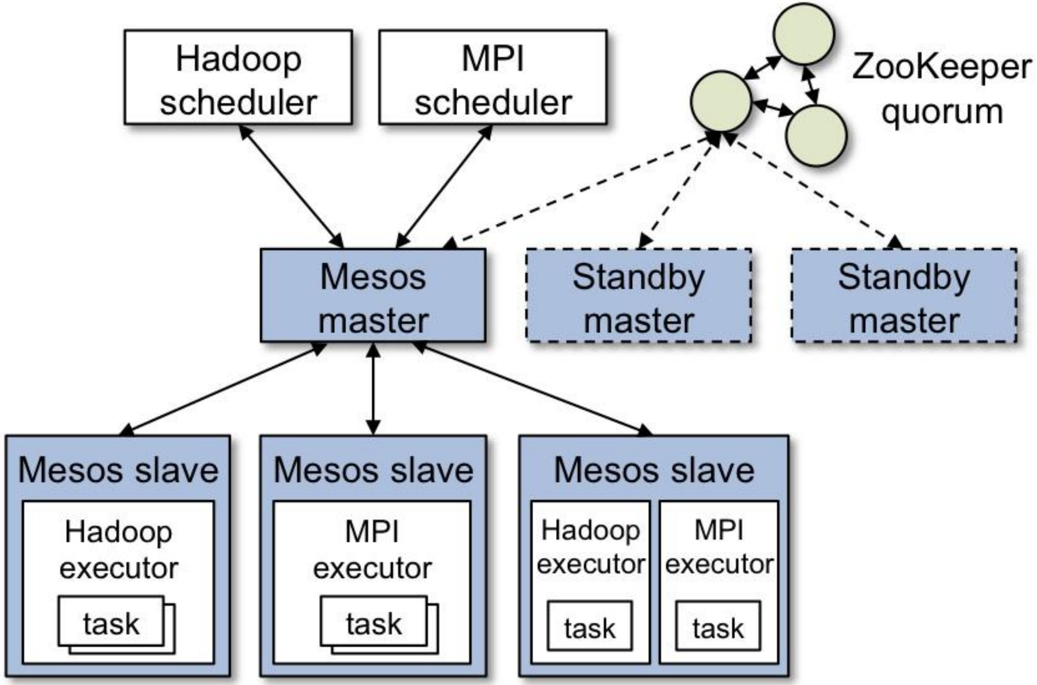
Mesos Master

- Manage Mesos slaves
- Make resource offer to frameworks



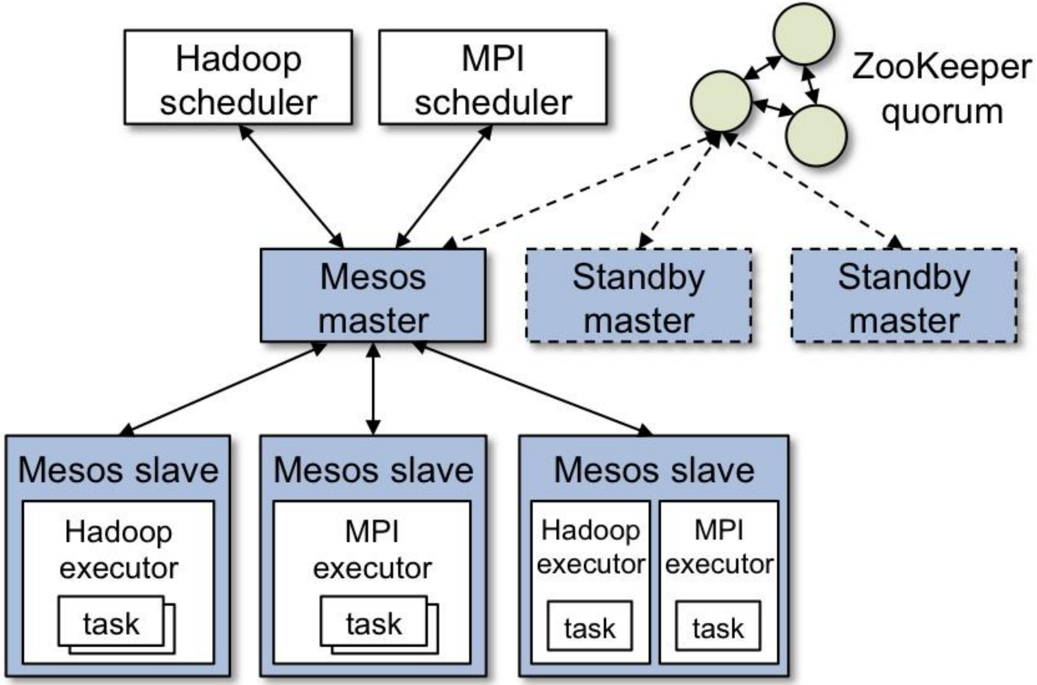
Mesos Slave

- Run tasks
- Report available resources
- Report task status, etc



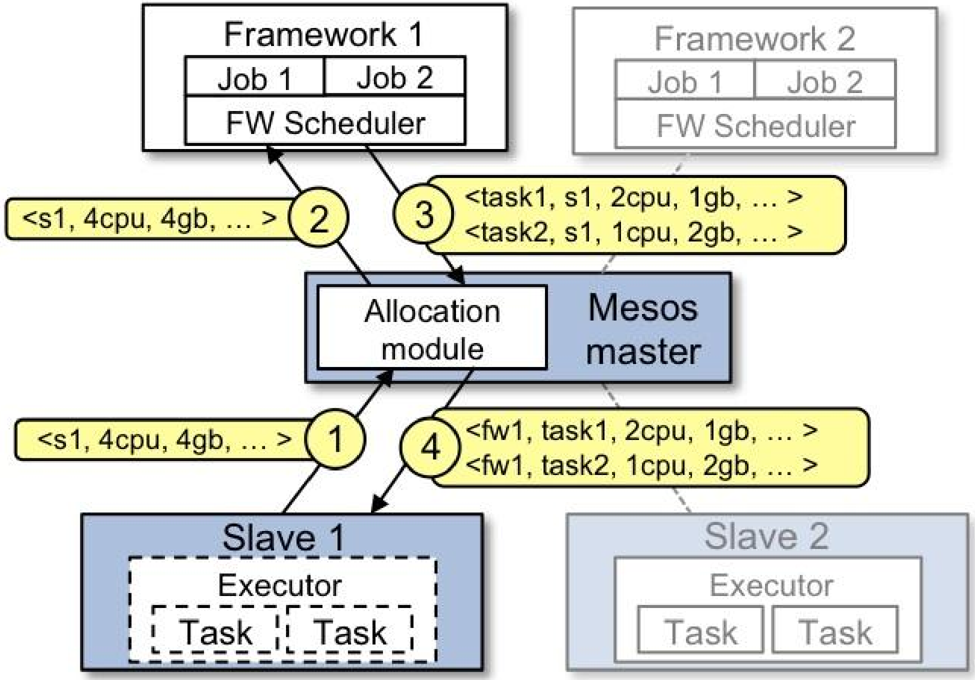
Mesos Framework

- Distributed Applications
- Scheduler processes offers and further schedule to tasks
- Executor launches tasks



Two Level Scheduling

- Allocation Module decide resources for each framework
- Framework Scheduler decide resources for each tasks



Scheduling Algorithm

- Dominant Resource Fairness Algorithm (DRF)
- Based on Benjemin Hinderman’s paper
- Default Mesos Allocation Module implementation
- Achieves fairness (maybe too much?)

An Example Problem

- Consider we have one pizza, 10 slices
- A want 1 slice, B want 2 slices, C want 5 slices, and D want 6 slices.
- How to achieve fairness in this case?
    - Divide equally - everyone get 2.5 slices, but C will get hungry
    - What if someone lied about their resource request?

Max-min Fairness Algorithm

- Satisfy small tasks first
- Divide the rest resources among others
  - A wants 1 slice, get 1 slice, 9 slices left
  - B wants 2 slices, get 2 slices, 7 slices left
  - C wants 5 slices, not enough left, divide with other people, get 3.5 slices, 3.5 slices left
  - D get 3.5 slices
- Max-min Fairness is a good solution for fairness

Dominant Resource Fairness (DRF)

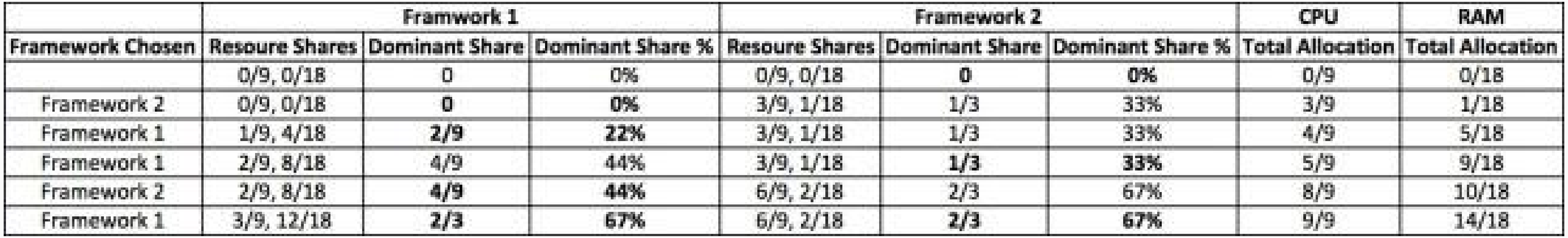
- DRF is a max-min fair algorithm for heterogeneous resources
  - CPU
  - Memory
  - IO

Dominant Resource Fairness (DRF)

- Nice qualities for scheduling algorithms
    - If C gets less than 2.5 slices, he will just go and buy his own pizza - Sharing Incentive
    - B lies about his share, what would happen? - Strategy Proofness
    - C should not envy B share - Envy Freeness
    - You cannot give B more pizza without give other people less pizza - Pareto Efficiency

Dominant Resource Fairness (DRF)

- Dominant Resource Share
  - If we have a system of 10 CPU 10 GB RAM, User A has been assigned with 2 CPU and 6 GB RAM, Dominant Resource Share is ⅗, and A's dominant resource is memory.
- DRF tries to maximize min dominant resource share.



	Framwork 1			Framework 2			CPU	RAM
Framework Chosen	**Resoure Shares**	**Dominant Share**	**Dominant Share %**	**Resoure Shares**	**Dominant Share**	**Dominant Share %**	**Total Allocation**	**Total Allocation**
	0/9, 0/18	0	0%	0/9, 0/18	**0**	**0%**	0/9	0/18
Framework 2	0/9, 0/18	**0**	**0%**	3/9, 1/18	1/3	33%	3/9	1/18
Framework 1	1/9, 4/18	**2/9**	**22%**	3/9, 1/18	1/3	33%	4/9	5/18
Framework 1	2/9, 8/18	4/9	44%	3/9, 1/18	**1/3**	**33%**	5/9	9/18
Framework 2	2/9, 8/18	**4/9**	**44%**	6/9, 2/18	2/3	67%	8/9	10/18
Framework 1	3/9, 12/18	**2/3**	**67%**	6/9, 2/18	**2/3**	**67%**	9/9	14/18

Agenda

- Use Cases
- What is Mesos
- Architecture
- **Mesos Usage**

Mesos Usage

- Shared Resource Pool
- Container Orchestration

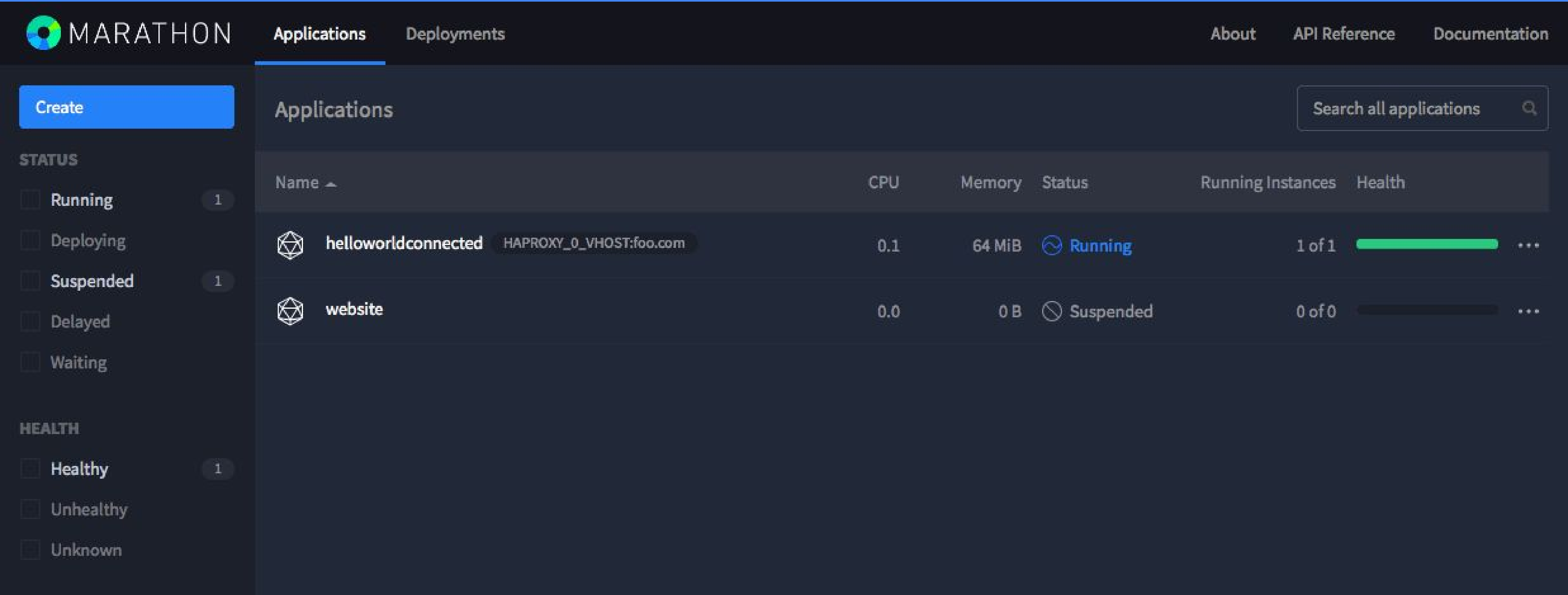
Shared Resource Pool

- Run mix of short tasks on Mesos
  - Jenkins - build jobs, deploy jobs
  - Spark
  - Hadoop
  - Ad-hoc queries
- Apple runs Siri on top of Mesos

Container Orchestration

- Use Apache Marathon (https://github.com/mesosphere/marathon)
  - High availability, failed containers will be respawned
  - Automatically load balancing
  - Service discovery

Container Orchestration



Applications

Name	CPU	Memory	Status	Running Instances	Health
helloworldconnected HAPROXY_0_VHOST:foo.com	0.1	64 MiB	Running	1 of 1	🟩
website	0.0	0 B	Suspended	0 of 0	

Similar Systems

- YARN (https://hortonworks.com/apache/yarn)
    - Developed to improve Hadoop resource efficiency
- Kubernetes (http://kubernetes.io/)
    - Container orchestration solution from Google
    - Based on Borg targeting Omega
- Docker Swarm (https://docs.docker.com/swarm/)
    - Native Container orchestration solution from Docker

Further Reading

- Borg: http://research.google.com/pubs/pub41684.html
- Borg, Omega, and Kubernetes: http://research.google.com/pubs/pub44843.html
- Omega: http://research.google.com/pubs/pub43438.html
- Mesos: https://www.cs.berkeley.edu/~alig/papers/mesos.pdf
- DRF: https://www.cs.berkeley.edu/~alig/papers/drf.pdf
- Orchestration: https://en.wikipedia.org/wiki/Orchestration_(computing)

Q&A