Big Data
- Apache YARN

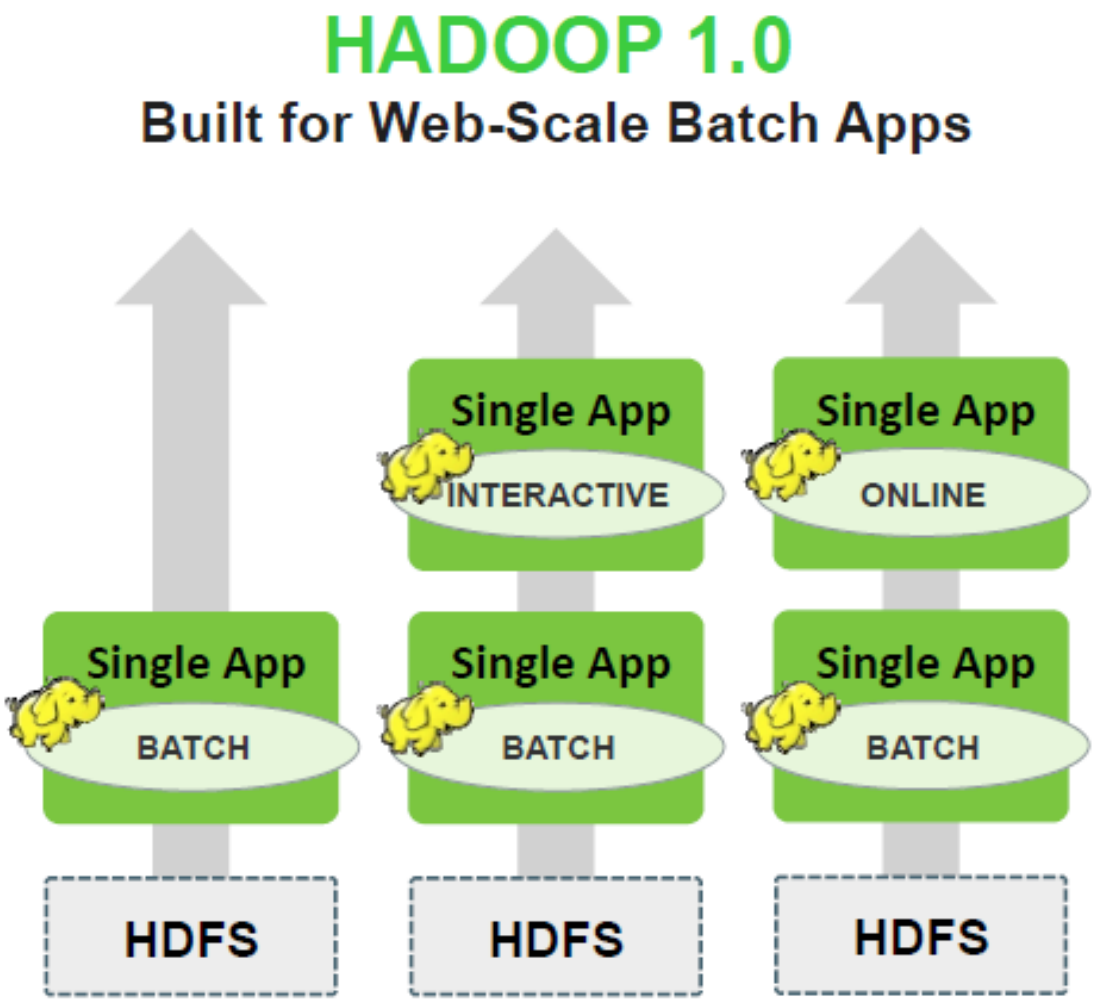
Dr. Matt Zhang
ITU

Outline

- Introduction to YARN
- YARN Architecture
- Implementing Applications

Introduction

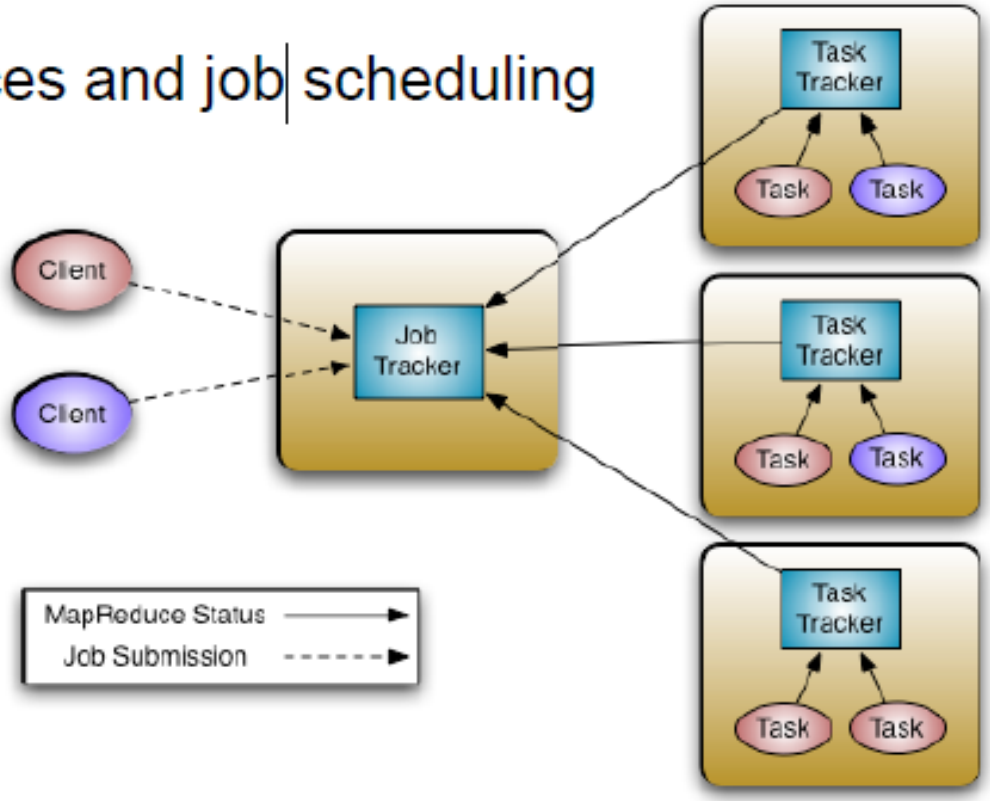
The 1st Gen Hadoop: Batch



- All other usage patterns must leverage that same infrastructure
- Forces the creation of silos for managing mixed workloads

Hadoop MapReduce Classic

- **JobTracker**
  - Manages cluster resources and job scheduling
- **TaskTracker**
  - Per-node agent
  - Manage tasks



The Problem

- Hadoop is being used for all kinds of tasks beyond its original design
- Tight coupling of a specific programming model with the resource management infrastructure
- Centralized handling of jobs' control flow

MapReduce Classic: Limitations

- **Scalability**
  - Maximum Cluster size – 4,000 nodes
  - Maximum concurrent tasks – 40,000
  - Coarse synchronization in JobTracker
- **Availability**
  - Failure kills all queued and running jobs

MapReduce Classic: Limitations

- **Hard partition of resources into map and reduce slots**
  - Low resource utilization
- **Lacks support for alternate paradigms and services**
  - Iterative applications implemented using MapReduce are 10x slower

Earlier versions

- **Scalability**
  - No.1 requirement from early versions to YARN
- Private Clusters
  - Desk application running on a host of machines
- Ad Hoc Clusters
  - Shared HDFS instance
  - Obtained security for multi-user environment
  - Setting up a shared cluster is nontrivial

Hadoop on Demand

- **Multi-tenancy**
  - A shared pool of nodes for all jobs
  - Allocate Hadoop clusters of fixed size on the shared pool.
- **Serviceability**
  - sets up a new cluster for every job
  - old and new Hadoop co-exist
  - Hadoop has short, 3-month release cycle

Failure of Hadoop on Demand

- Scalability
- Multi-tenancy
- Serviceability
- **Locality Awareness**
  - JobTracker tries to place tasks close to the input data
  - But node allocator is not aware of the locality

Failure of Hadoop on Demand

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- **High Cluster Utilization**
  - HoD does not resize the cluster between stages
  - Users allocate more nodes than needed
    - Competing for resources results in longer latency to start a job

Problem with Shared Cluster

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- **Reliability/Availability**
  - The failure in one job tracker can bring down the entire cluster
  - Overhead of tracking multiple jobs in a larger, shared cluster

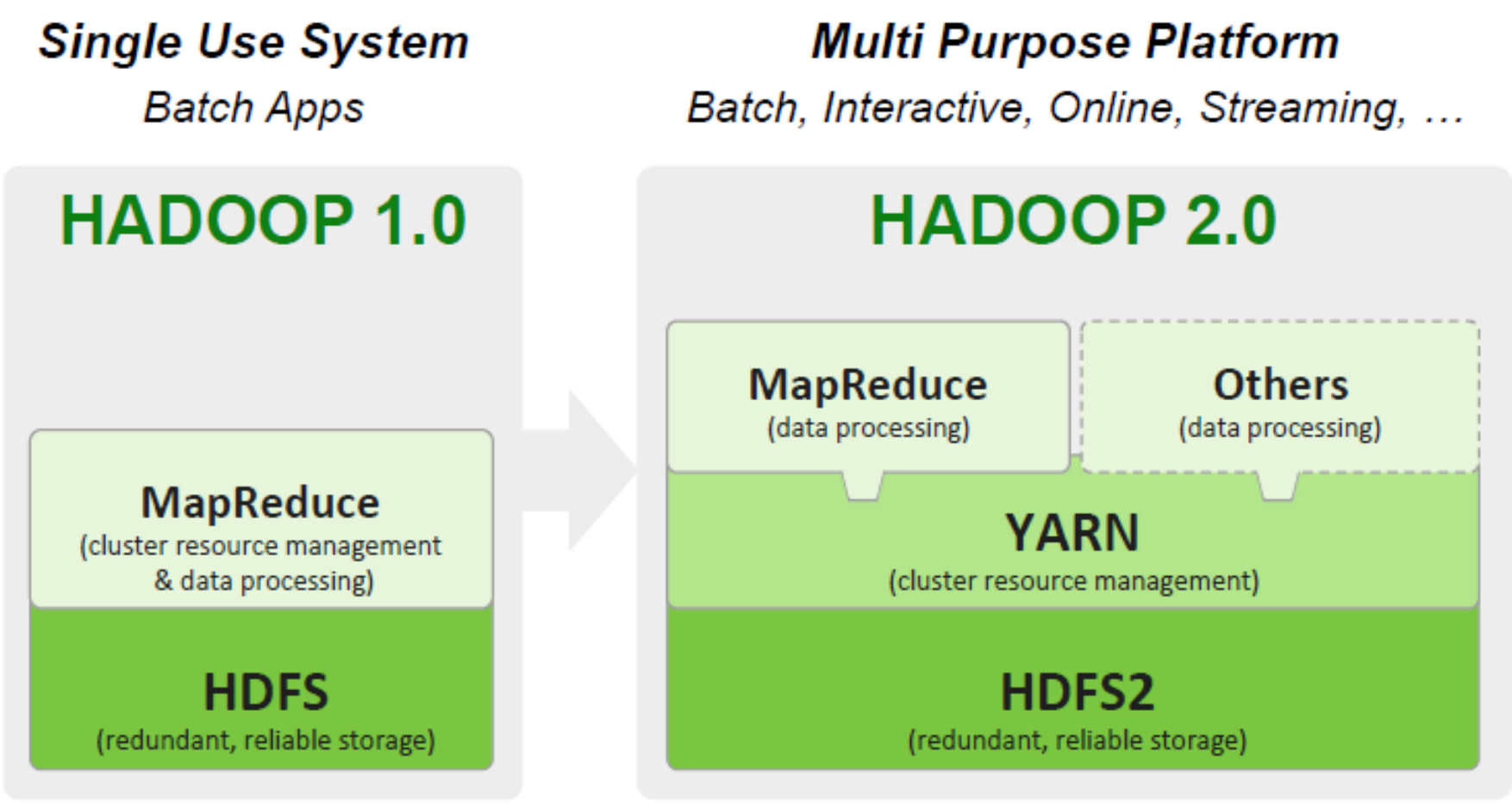
Challenge of Multi-tenancy

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- **Secure and auditable operation**
  - Authentication
  - Auditable usage of cluster resources

Challenge of Multi-tenancy

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- Secure and auditable operation
- **Support for Programming Model Diversity**
  - Iterative computation
  - Different communication pattern

Hadoop as Next-Gen Platform



YARN

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- Secure and auditable operation
- Support for Programming Model Diversity

- **Flexible Resource Model**
  - Hadoop: # of Map/reduce slots are fixed.
  - Easy, but lower utilization

YARN

- Scalability
- Multi-tenancy
- Serviceability
- Locality Awareness
- High Cluster Utilization
- Reliability/Availability
- Secure and auditable operation
- Support for Programming Model Diversity
- Flexible Resource Model

- **Backward Compatibility**
  - The system behaves similar to the old Hadoop

YARN

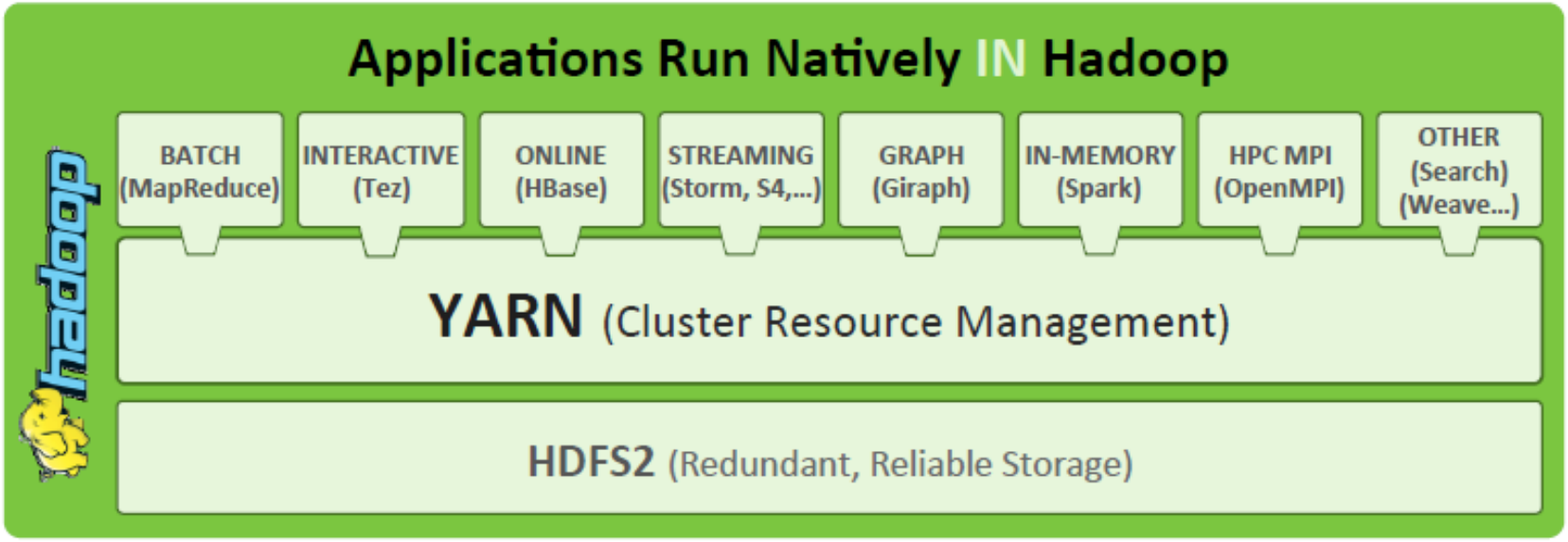
- Separating resource management functions from the programming model
- MapReduce becomes just one of the application
- Binary compatible/Source compatible

YARN: Taking Hadoop Beyond Batch

Store ALL DATA in one place…

Interact with that data in MULTIPLE WAYS

with Predictable Performance and Quality of Service



5 Key Benefits of YARN

1. Scale
2. New Programming Models & Services
3. Improved cluster utilization
4. Agility
5. Beyond Java

Yarn Architecture

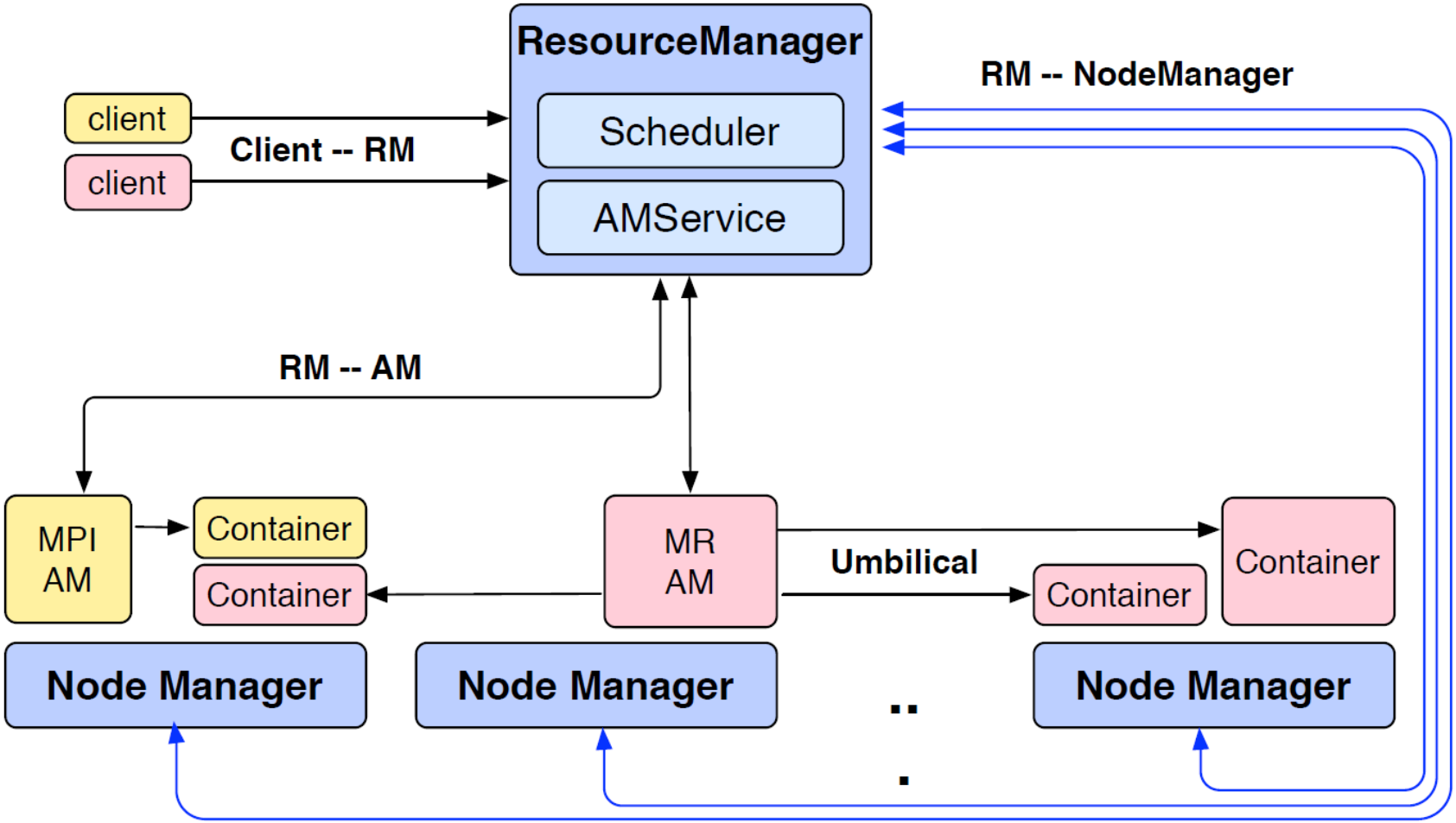
A Brief History of YARN

- **Originally conceived & architected by the team at Yahoo!**
  - Arun Murthy created the original JIRA in 2008 and led the PMC
- **The team at Hortonworks has been working on YARN for 4 years**
- **YARN based architecture running at scale at Yahoo!**
  - Deployed on 35,000 nodes for 6+ months
- **Multitude of YARN applications**

Concepts

- Application
  - Application is a job submitted to the framework
  - Example – Map Reduce Job
- Container: Replaces the fixed map/reduce slots
  - Basic unit of allocation
  - Fine-grained resource allocation across multiple resource types (memory, cpu, disk, network, ..)
    - container_0 = 2GB, 1CPU
    - container_1 = 1GB, 6 CPU

YARN: Architecture



Resource Manager

- One per cluster
  - Central, global view
  - Enable global properties
    - Fairness, capacity, locality
- Container
  - Logical bundle of resources (CPU/memory)
- Job requests are submitted to RM
    - To start a job, RM finds a container to spawn AM
- No static resource partitioning

Resource Manager (cont')

- only handles an overall resource profile for each application
  - Local optimization/internal flow is up to the application
- Preemption
  - Request resources back from an application
  - Checkpoint snapshot instead of explicitly killing jobs / migrate computation to other containers

Application Master

- The head of a job
- Runs as a container
- Request resources from RM
  - # of containers/ resource per container/ locality …
- Dynamically changing resource consumption
- Can run any user code (Dryad, MapReduce, Tez, REEF…etc)
- Requests are "*late-binding*"

MapReduce AM

- Optimizes for locality among map tasks with identical resource requirements
  - Selecting a task with input data close to the container.
- AM determines the semantics of the success or failure of the container

Node Manager

- The “worker” daemon. Registers with RM
- One per node
- Container Launch Context – env. var, commands…
- Report resources (memory/CPU/etc…)
- Configure the environment for task execution
- Garbage collection/ Authentication
- Auxiliary services
  - Output intermediate data between map and reduce tasks

YARN framework/application writers

1. Submitting the application by passing a Container Launch Context(CLC) for the Application Master to the RM.
2. When RM starts the AM, it should register with the RM and periodically advertise its liveness and requirements over the heartbeat protocol

YARN framework/application writers

3. Once the RM allocates a container, AM can construct a CLC to launch the container on the corresponding NM. It may also monitor the status of the running container and stop it when the resource should be reclaimed. Monitoring the progress of work done inside the container is strictly the AM's responsibility.

YARN framework/application writers

4. Once the AM is done with its work, it should unregister from the RM and exit cleanly.
5. Optionally, framework authors may add control flow between their own clients to report job status and expose a control plane.

Fault tolerance and availability

- RM Failure
  - Recover using persistent storage
  - Kill all containers, including AMs'
  - Re-launch AMs
- NM Failure
  - RM detects it, mark the containers as killed, report to AMs
- AM Failure
  - RM kills the container and restarts it.
- Container Failure
  - The framework is responsible for recovery

Implementing Applications

YARN – Implementing Applications

- What APIs do I need to use?
  - Only three protocols
    - Client to ResourceManager
      - Application submission
    - ApplicationMaster to ResourceManager
      - Container allocation
    - ApplicationMaster to NodeManager
      - Container launch

YARN – Implementing Applications

- Use client libraries for all 3 actions
  - Module yarn-client
  - Provides both synchronous and asynchronous libraries
  - Use 3rd party like Weave
    - http://continuuity.github.io/weave/

YARN – Implementing Applications

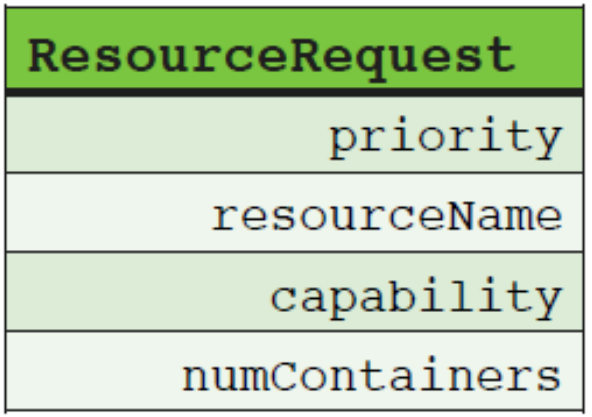
- What do I need to do?
  - Write a submission Client
  - Write an ApplicationMaster
  - Get containers, run whatever you want!

YARN – Implementing Applications

- **What else do I need to *know*?**
  - Resource Allocation & Usage
    - ResourceRequest
    - Container
    - ContainerLaunchContext
    - LocalResource
  - ApplicationMaster
    - ApplicationId
    - ApplicationAttemptId
    - ApplicationSubmissionContext

Resource Allocation & Usage

- ResourceRequest
  - Fine-grained resource ask to the ResourceManager
  - Ask for a specific amount of resources (memory, cpu etc.) on specific machine or rack
  - Use special value of * for resource name for any machine



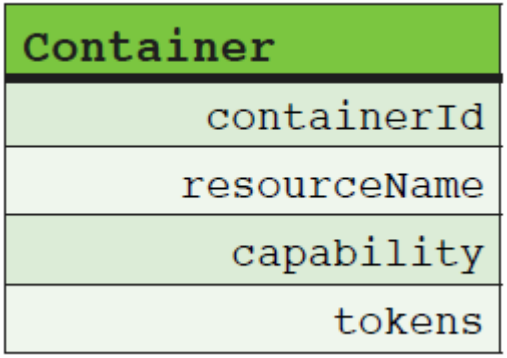
Resource Allocation & Usage

- ResourceRequest

priority	capability	resourceName	numContainers
0	<2gb, 1 core>	host01	1
		rack0	1
		*	1
1	<4gb, 1 core>	*	1

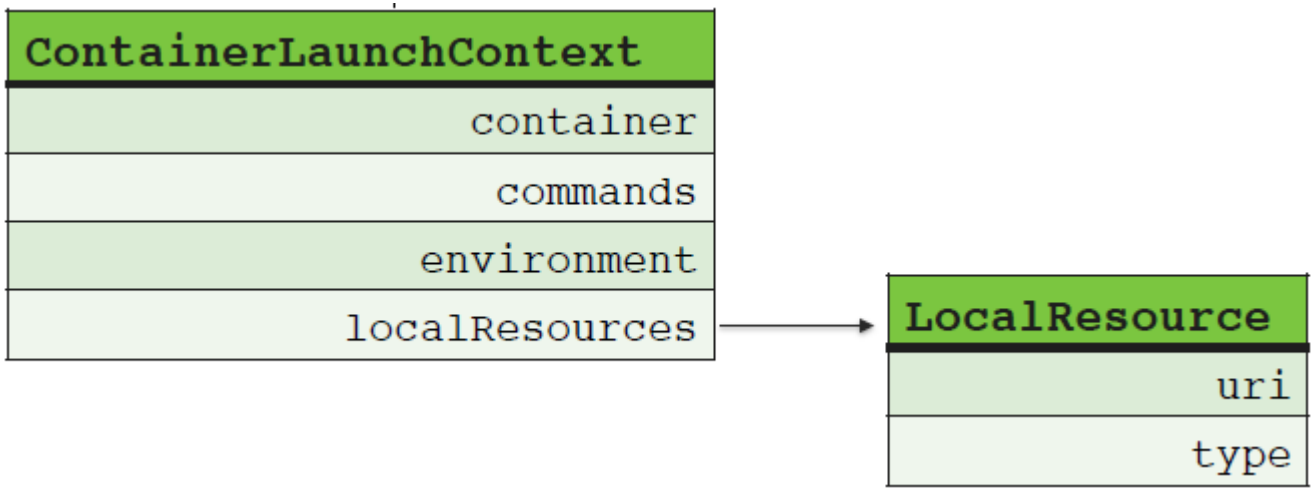
Resource Allocation & Usage

- Container
  - The basic unit of allocation in YARN
  - The result of the ResourceRequest provided by ResourceManager to the ApplicationMaster
  - A specific amount of resources (cpu, memory etc.) on a specific machine



Resource Allocation & Usage

- **ContainerLaunchContext**
  - The context provided by ApplicationMaster to NodeManager to launch the Container
  - *Complete specification for a process*
  - LocalResource used to specify container binary and dependencies
    - NodeManager responsible for downloading from shared namespace (typically HDFS)

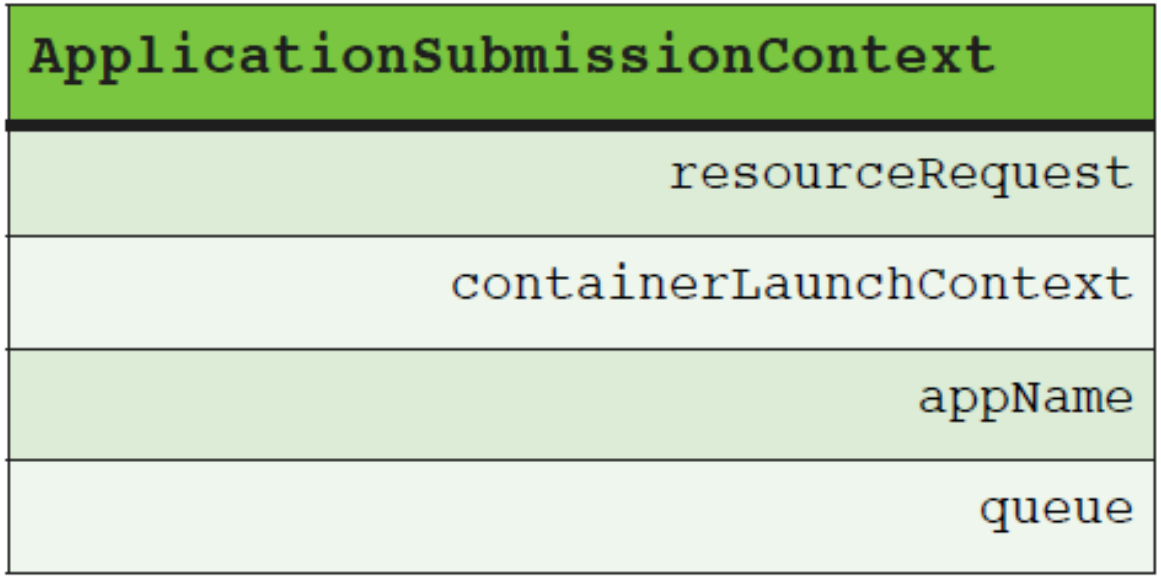


ApplicationMaster

- Per-application controller aka container_0
- Parent for all containers of the application
  - ApplicationMaster negotiates all its containers from ResourceManager
- ApplicationMaster container is child of ResourceManager
  - RM restarts the ApplicationMaster attempt if required (unique ApplicationAttemptId)
- Code for application is submitted along with Application itself

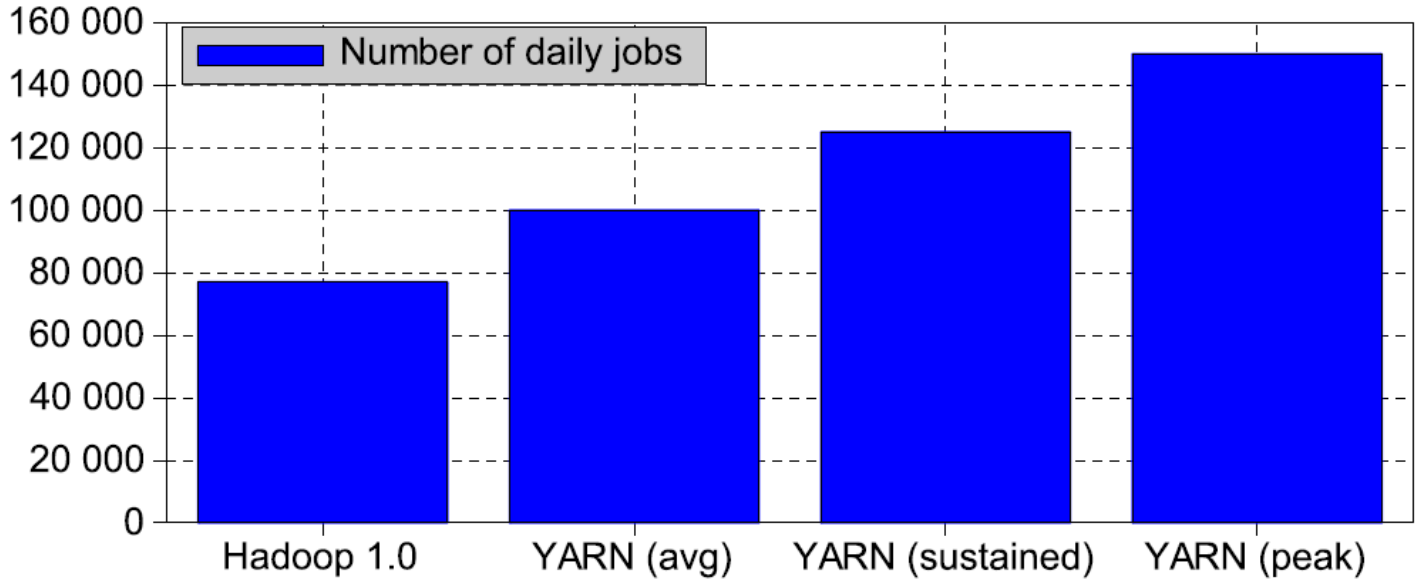
ApplicationMaster

- ApplicationSubmissionContext is the complete specification of the ApplicationMaster, provided by Client
- ResourceManager responsible for *allocating* and *launching* ApplicationMaster container



Example: YARN at Yahoo!

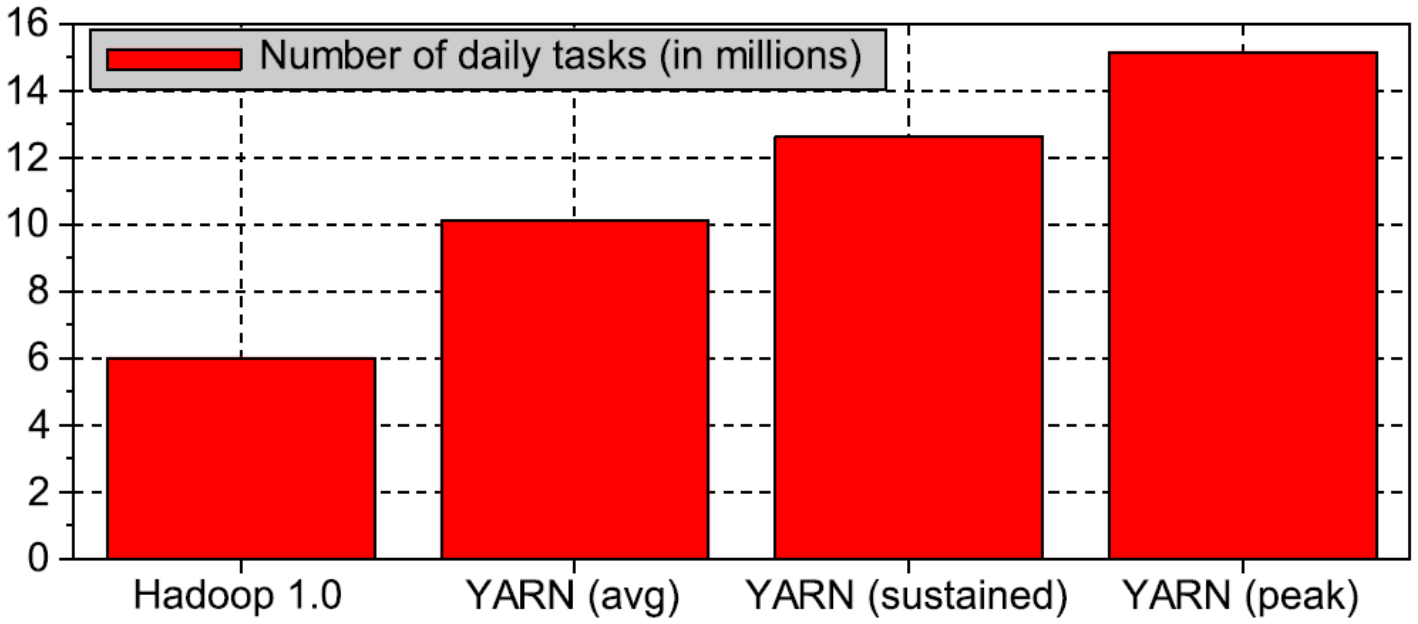
- In a 2500-node cluster, throughput improves from 77 K jobs/day to 150 K jobs/day



(a) Daily jobs

YARN at Yahoo!

- In a 2500-node cluster, throughput improves from 4 M tasks/day to 10 M jobs/day



(b) Daily tasks

YARN at Yahoo!

- Why?
  - the removal of the static split between map and reduce slots.
- Essentially, moving to YARN, the CPU utilization almost doubled
- **“upgrading to YARN was equivalent to adding 1000 machines [to this 2500 machines cluster]”**

Summary

Summary

- YARN Separates resource management functions from the programming model
- Resource Manager tracks resource usage and node liveness
- Upgrading to YARN equivalent to adding lots of machines