IERG4330
Programming Big Data Systems

Resource Management and Infrastructure for Big Data Systems and Cloud-Native Applications

Prof. Wing C. Lau
Department of Information Engineering
wclau@ie.cuhk.edu.hk

Acknowledgements

- The slides used in this chapter are adapted from the following sources:
  - "Data-Intensive Information Processing Applications," by Jimmy Lin, University of Maryland.

  - "Intro To Hadoop" in UCBerkeley i291 - Analyzing BigData with Twitter, by Bill Graham, Twitter.
  - Ryza of Cloudera Inc, "Can't we just get along ?", Spark Summit, 2013
  - Cloudera, "Introduction to YARN and MapReduce 2", SlideShare.net
  - Eric Brewer, Google VP of Instructure, "Google Tech Talk – Containers: What, Why, How ; Google Cloud Innovation", April 2015
  - Ajit Punj, Juan Manuel Camacho, Borg – a presentation for Stanford CS349d, Fall 2018
  - Alex Gilkson of CMU, "Cloud-Native Applications and Kubernetes (k8s), 2019
  - Cyberlearn CLOUD 2019-2020 (Master) MSE Lecture notes on Kubernetes

    https://cyberlearn.hes-so.ch/course/view.php?id=14014
  - Kubernauts – The Cloud Cosmonauts "The Kubernetes Learning Slides," v0.15.1, June 15, 2020.

    https://docs.google.com/presentation/d/13EQKZSQDounPC1I6EC4PmqaRmdCrpT3qswQJz9KRCyE/htmlpresent
  - Bob Killen, Cloud Native Computing Foundation (CNCF) Ambassador, "Kubernetes – an Introduction," July 2019.

    https://docs.google.com/presentation/d/1zrfVIE5r61ZNQrmXKx5gJmBcXnoa_WerHEnTxu5SMco/edit#slide=id.g3cfa019267_4_0
  - Alex Gilkson of CMU, "Cloud-Native Applications and Kubernetes (k8s), 2019

Recap:
Dataflow progamming with MapReduce

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

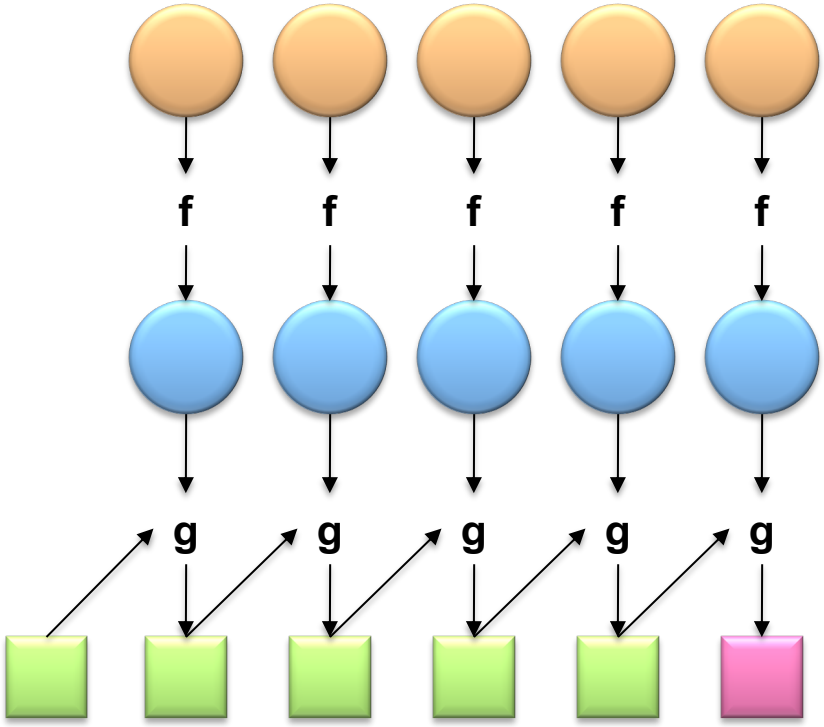
  reduce (k', v') → <k'', v''>*

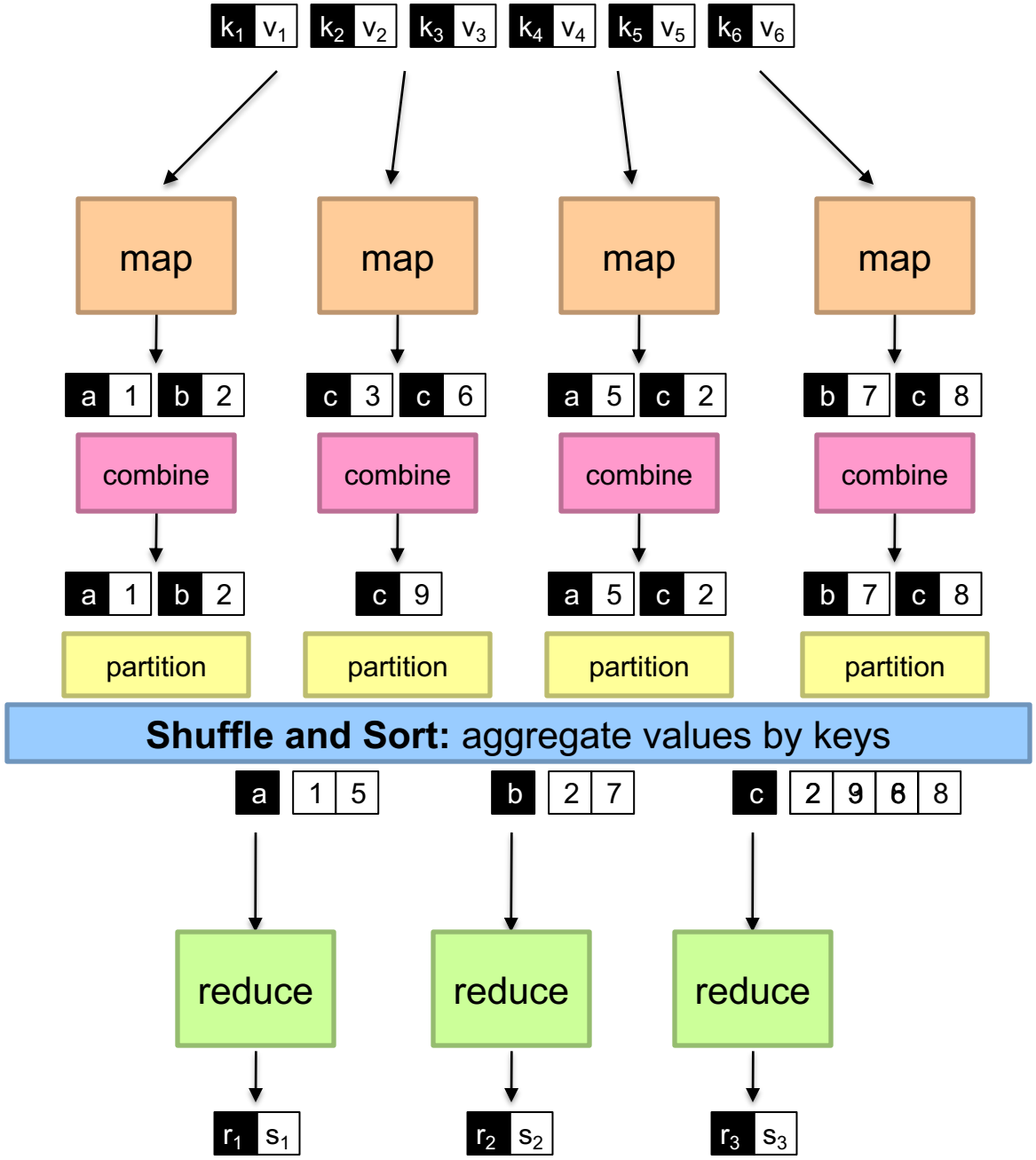
  - All values with the same key are sent to the same reducer
  - <a,b>* means a list of tuples in the form of (a,b)
- The execution framework handles everything else…

MapReduce:
A Dataflow Programming Model with Roots in Functional Programming

Map

Fold




k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
combine
combine
combine
combine
a 1
b 2
c 9
a 5
c 2
b 7
c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

MapReduce: A Dataflow Programming Model with Roots in Functional Programming

- What is the Advantage of adopting a "Dataflow" Model ?
- What is the Advantage of adopting a "Functional Programming" approach ?
  - There are NO Side Effects of Computation for PURE Functional Programming

=> Significantly Simplify Parallelization & Failure Recovery

Consider the following (non-functional) imperative programming example:

```
var x = 0 ;
async { x = x + 1 }
async { x = x * 2 }

// x can be 0, 1 or 2
```

References:
1. Kevin Hammond, "Why Parallel Functional Programming Matters: Panel Statement", Reliable Software Technologies, Ada-Europe 2011, LNCS Vol. 6652, 2011, *http://link.springer.com/book/10.1007/978-3-642-21338-0*
2. Martin Odersky, "Working Hard to Keep it Simple -- Why Functional Programming & Parallel-processing is a good fit," Keynote for OSCON Java 2011, *https://www.youtube.com/watch?v=3jg1AheF4n0*

But MapReduce is NOT good for…

- Jobs require multiple iterations and multiple-stages of operations
- Low-latency jobs
- Jobs that need shared state/ coordination
  - Tasks are shared-nothing
  - Shared-state requires scalable state store
- Jobs on small datasets
- Finding individual records

For some of these, we will introduce alternative computational models/ platforms, e.g. GraphLab, Spark, later in the course

Big Data Programming Models beyond MapReduce

- Many of them still takes the "Dataflow" programming model BUT generalize MapReduce by:

i) Relaxing the Rigid (fixed) structure of the Dataflow graph (topology) imposed by MapReduce, e.g.

  - Many can support Dataflow computations which can be expressed as a Directed Acyclic Graph (DAG), e.g. Dryad, Tez, Spark, Storm, etc.
  - More recent ones can even support computations which correspond to Stateful and Loopy Dataflow graphs, e.g. Naiad (from MSR) and Tensorflow (Google)

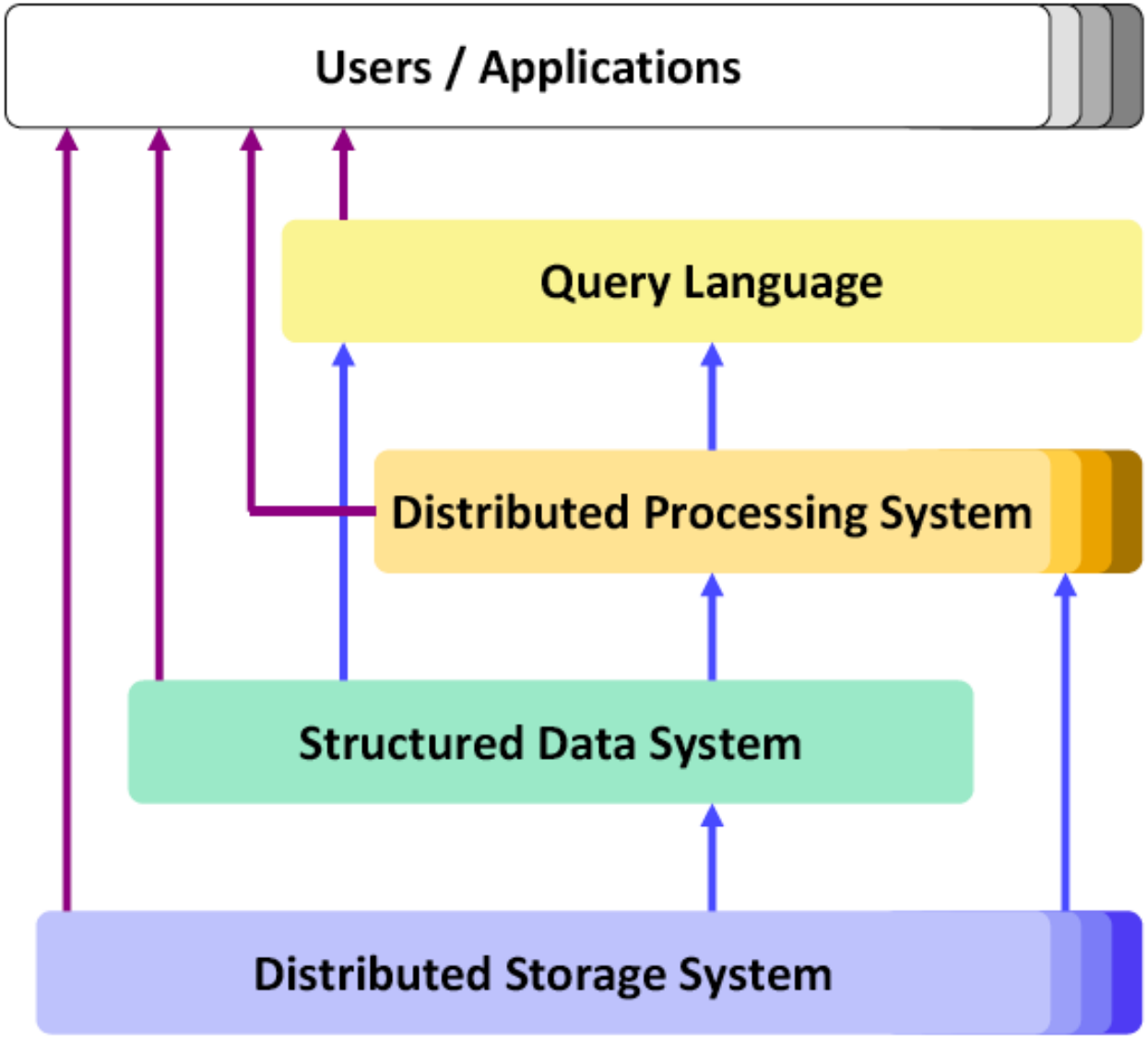
ii) Support Higher-level programming construct, e.g.

  - Use SQL-like query language, e.g. Hive and Spark SQL, and provide under-the-hood parallelization and optimization, by automatically transform the computation to some coordinated MapReduce job(s).
  - Create new Dataflow languages and systems which specify parallel operations/transformation on a distributed collections of (data) objects, e.g. LINQ, DryadLINQ, Pig-latin/Pig, Spark

- Other Big Data programming frameworks/ models for specific types of input data, e.g. to support Graph-based problems (e.g. GraphLab, Pregel) or Stream-based computation (Storm).

The Big Data Processing Stack:

Typical Architecture: Different Component Systems for various Services and Functionalities



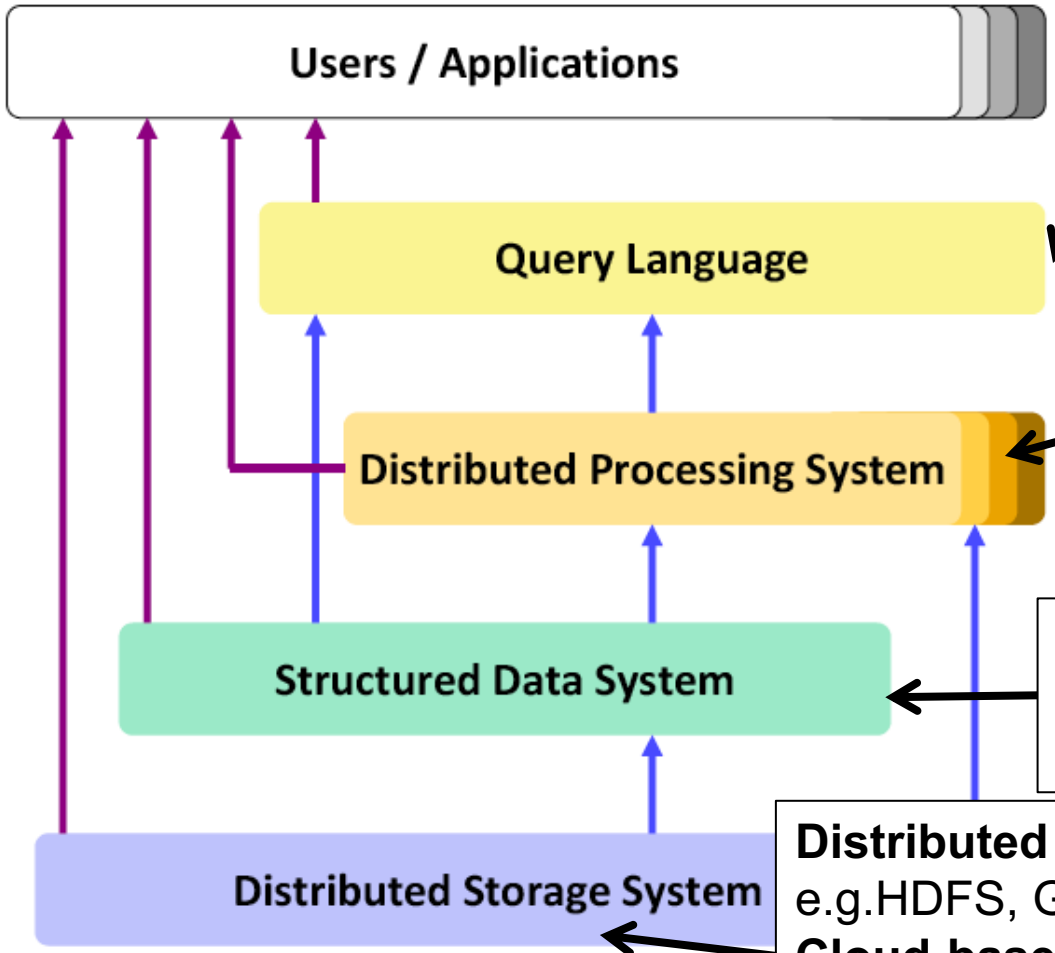
High-level language for accessing data and controlling processing, but typically not SQL!

Performance for complex operations (SQL-style joins and grouping, data analysis, etc.)

Simple but flexible data model (mostly key/value pairs), basic access operations (lookup-API)

Performance for data accesses, fault-tolerance, availability, scalability

Typical Architecture: Different Component Systems for various Services and Functionalities



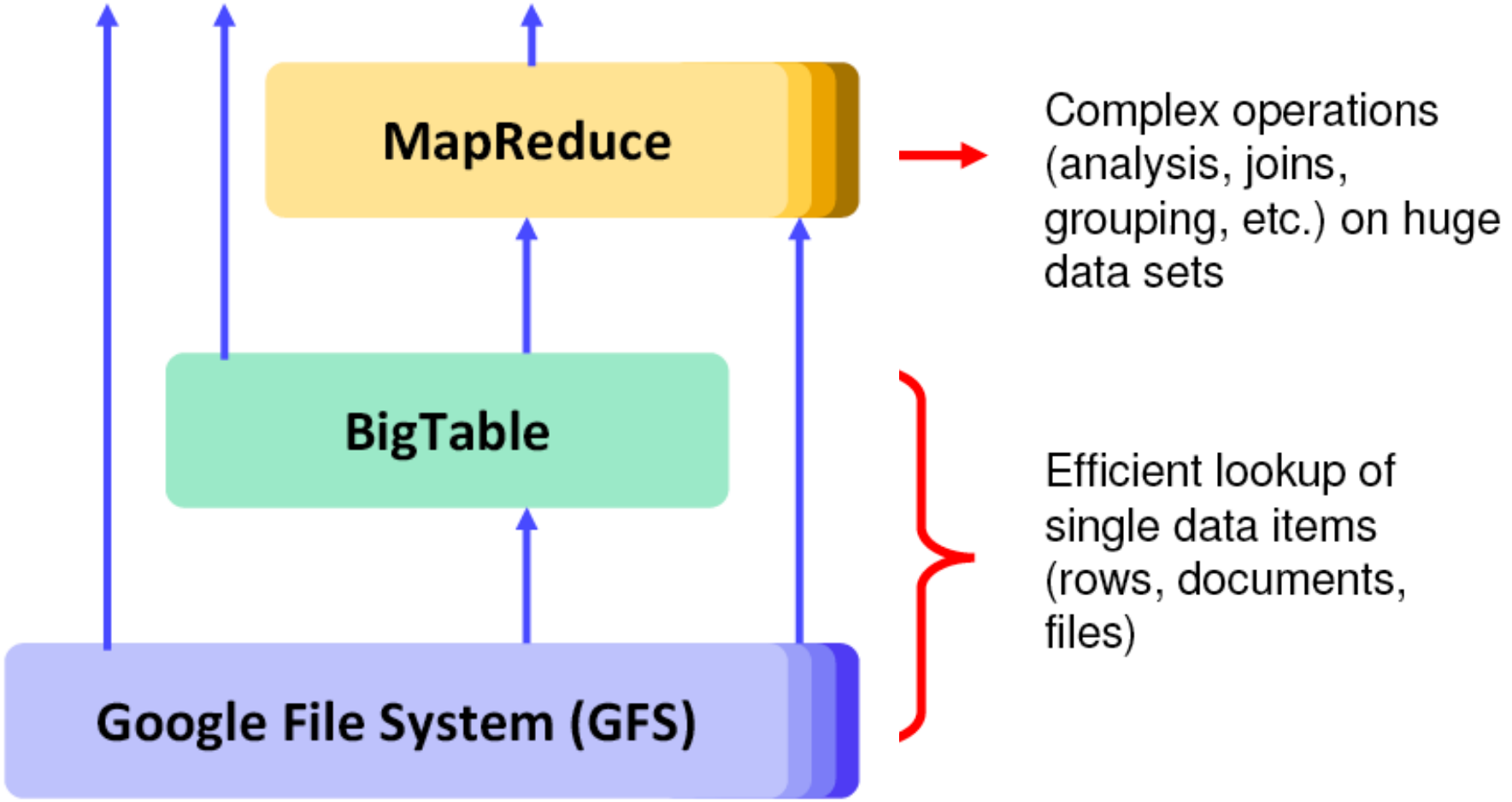
e.g. HiveQL of Hive (Facebook), BigSQL (IBM), Apache Drill, Cloudera Impala ; Pig (Yahoo); Spark SQL, DryadLINQ, other NoSQL query languages (NoSQL = Not-only-SQL)

e.g. Hadoop/ MapReduce, GraphLab (CMU/UWash), Spark (Berkeley), Storm/Heron (Twitter), Dryad (Microsoft), TeZ, Pregel/ Giraph (Apache), Flink

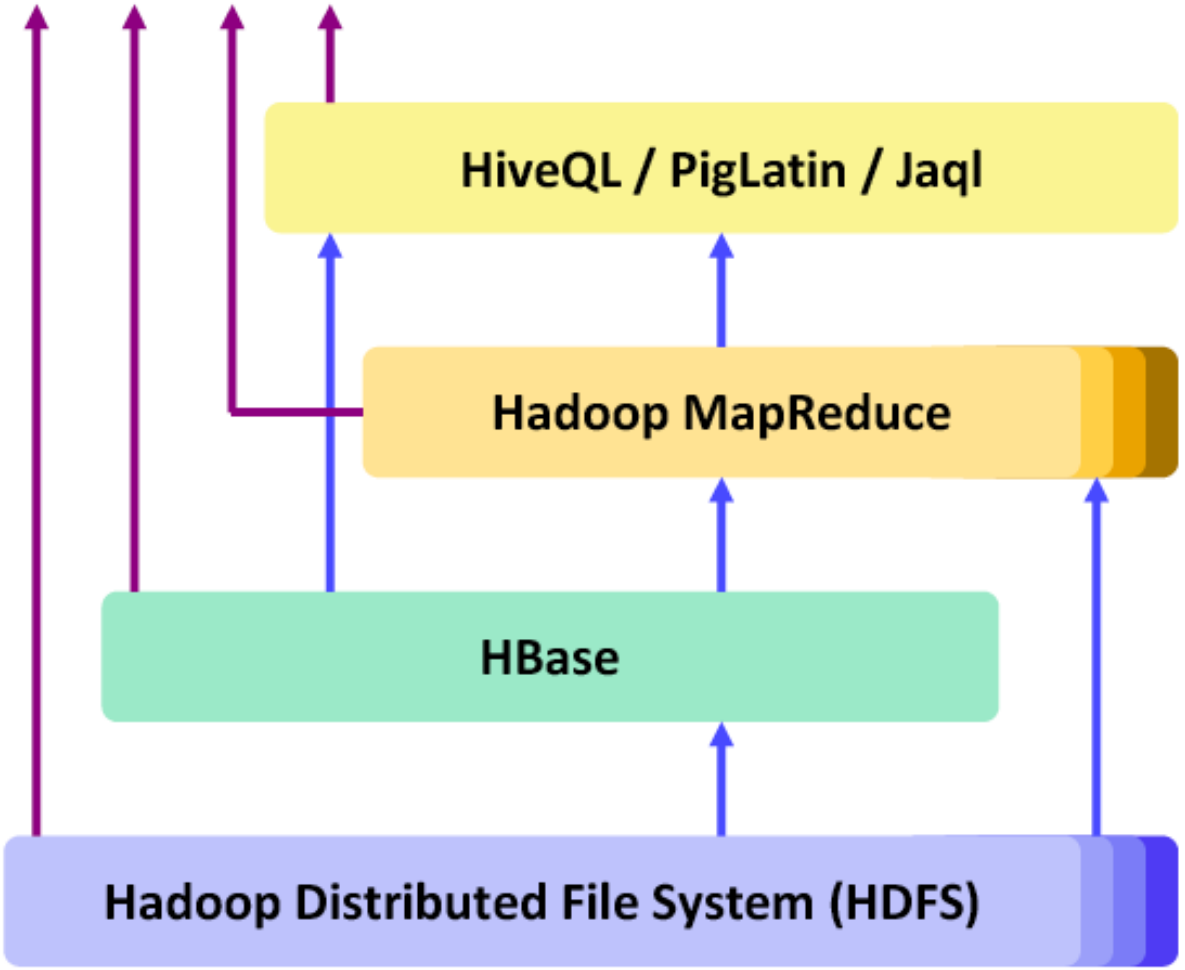
e.g. BigTable(Google)/ Hbase(open) SimpleDB, DynamoDB (Amazon), Cassandra(Facebook)

Distributed FileSystems:
e.g.HDFS, GFS, ceph
Cloud-based Data-Store Service/System:
e.g. Amazon S3, EBS, OpenStack Swift , Amazon Dynamo <key,value> store

Architecture Sample 1: The (old) Google-way (circa 2004)

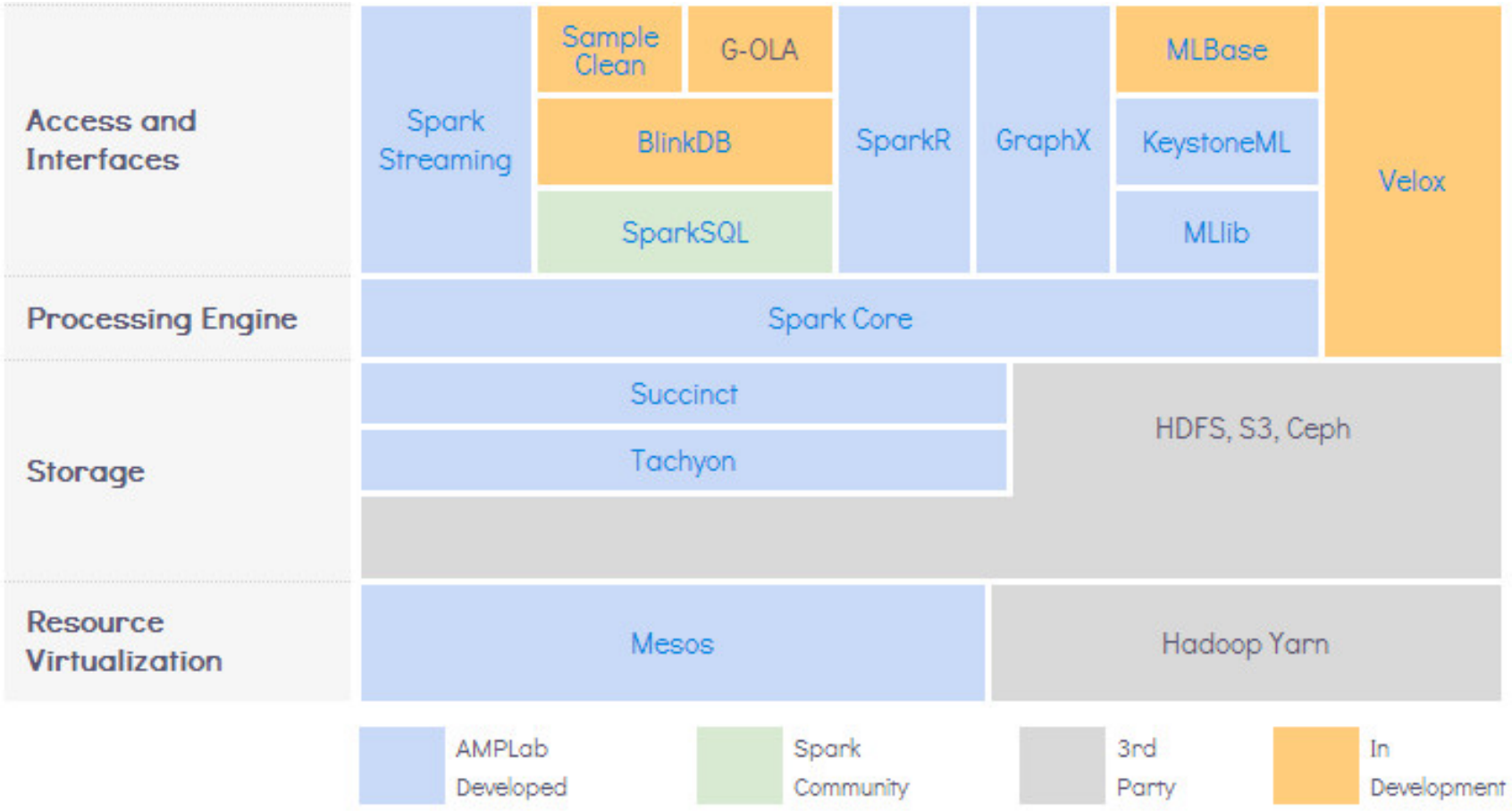


Architecture Sample 2: The Hadoop-way (e.g. Yahoo)
(circa 2007)



Beyond Hadoop/MapReduce: Another Main-stream Big Data Processing Framework

- Spark & Big (Berkeley) Data Analytic Stack (BDAS) by UC Berkeley



Reference: https://amplab.cs.berkeley.edu/software/

Recap:

Runtime Support & Resource Management for MapReduce/ Hadoop 1.0

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k'', v''>*

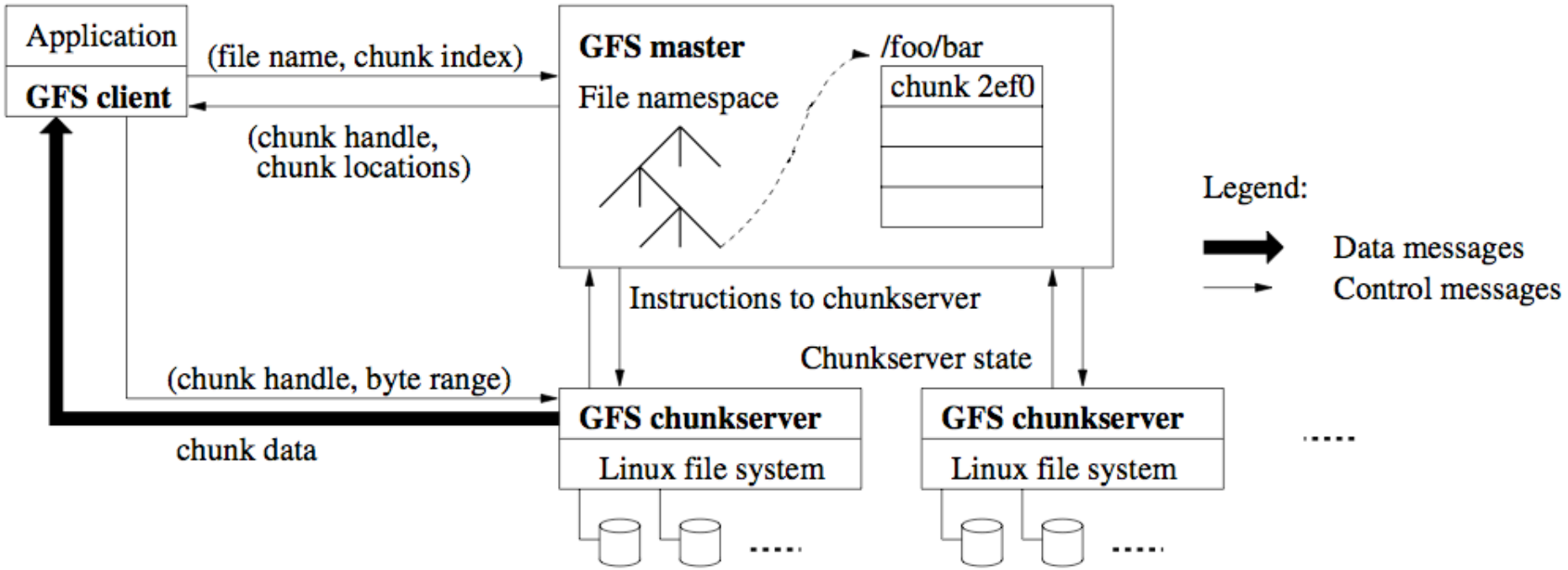
  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What's "everything else"?

The MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of the distributed Google File System (or HDFS)

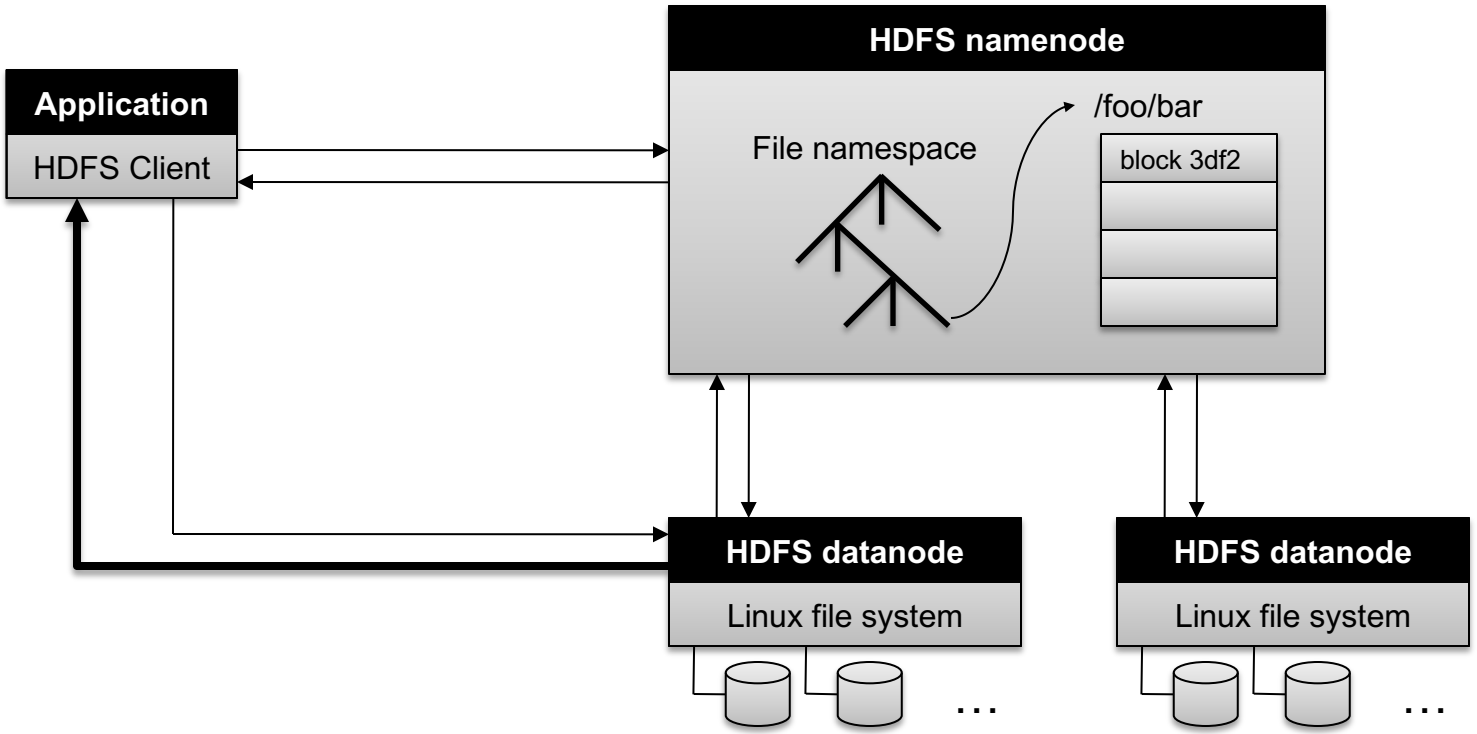
Google File System



Ghemawat, Gobioff, Leung, 2003

- Chunk servers hold blocks of the file (64MB per chunk)
- Replicate chunks (chunk servers do this autonomously). More bandwidth and fault tolerance
- Master distributes, checks faults, rebalances (Achilles heel)
- Client can do bulk read / write / random reads

HDFS Architecture

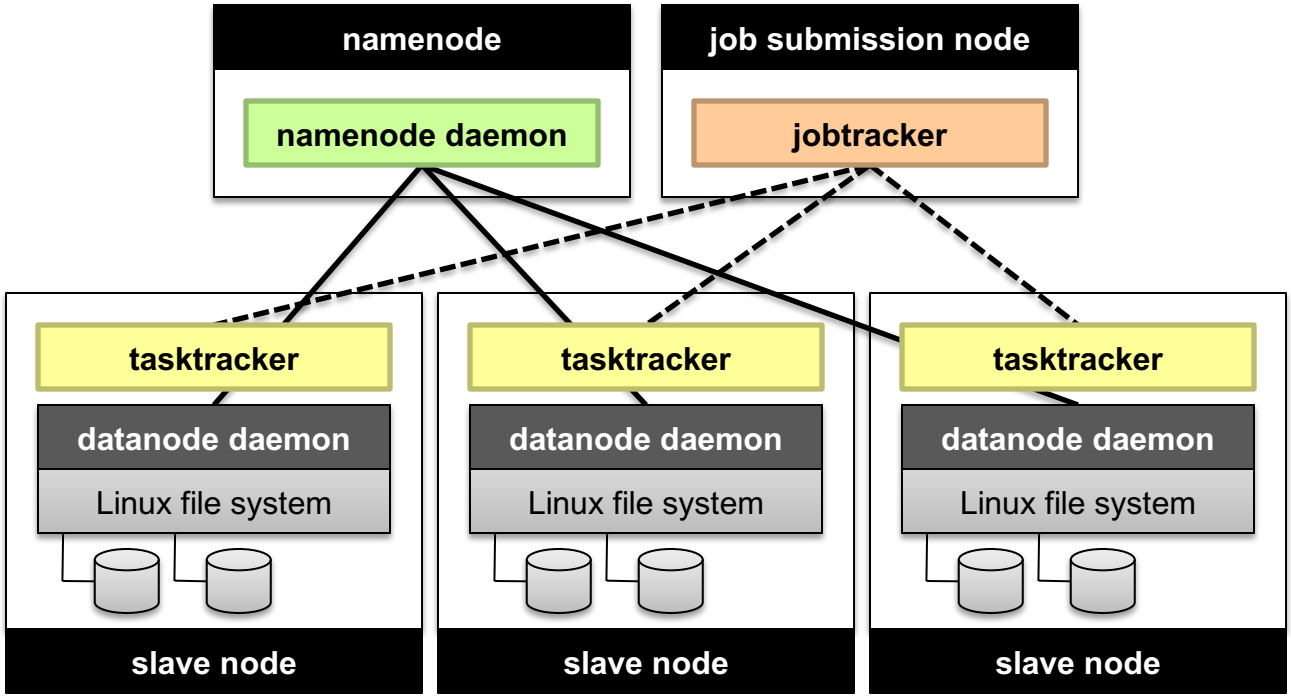


Adapted from (Ghemawat et al., SOSP 2003)

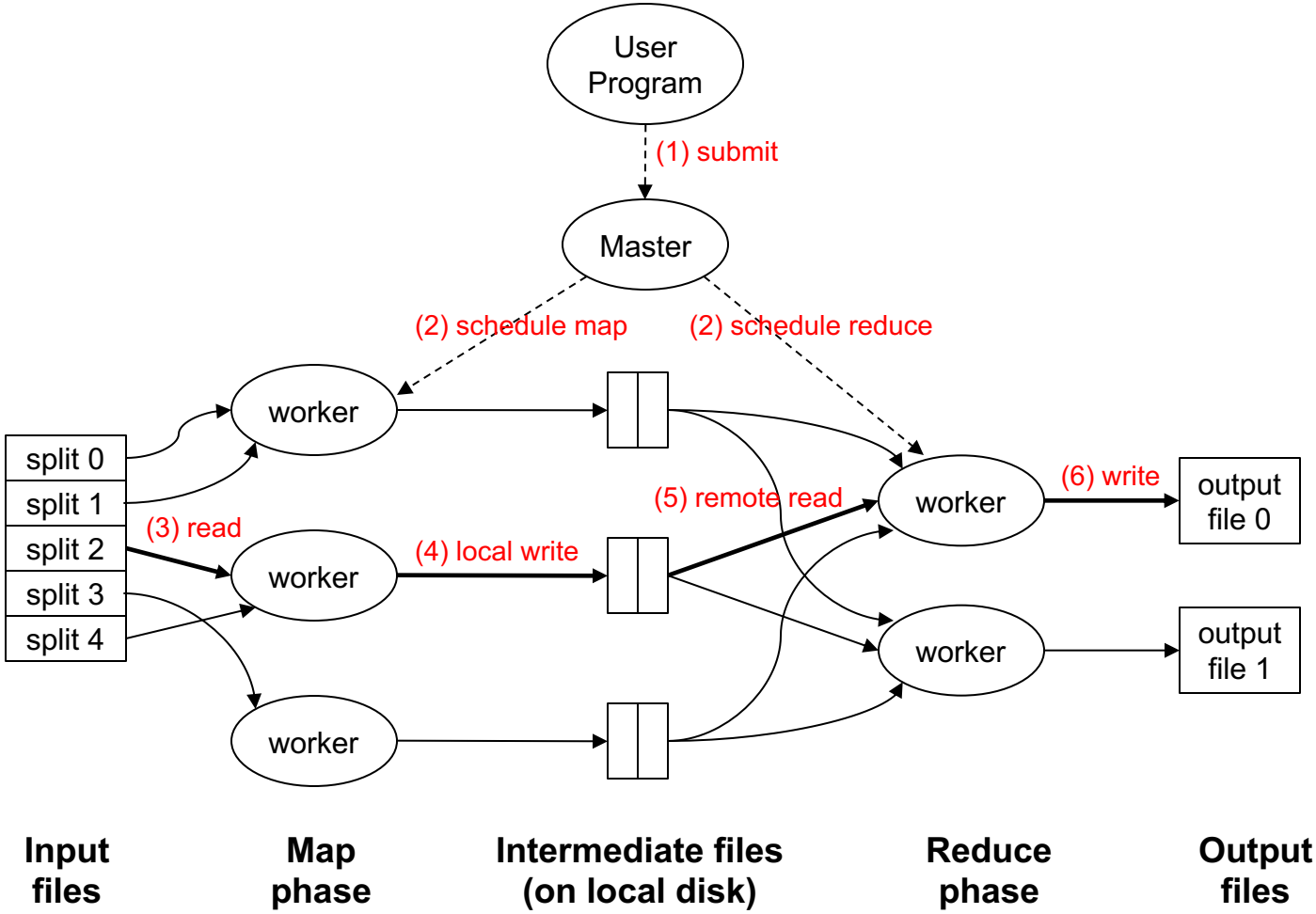
Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection
- Namenode can be Archille's heel – Single point of failure or bottleneck of scalability for the entire FS:
  - Need to have a Backup Namenode HDFS (or Master in GFS)
  - Compared to the fully-distributed approach in Ceph

Putting everything together…



Job Scheduling for MapReduce/Hadoop 1.0



Adapted from (Dean and Ghemawat, OSDI 2004)

Resource Management Platforms for Big Data Processing Clusters

Hadoop 1.0 vs. Hadoop 2.0 Ecosystem

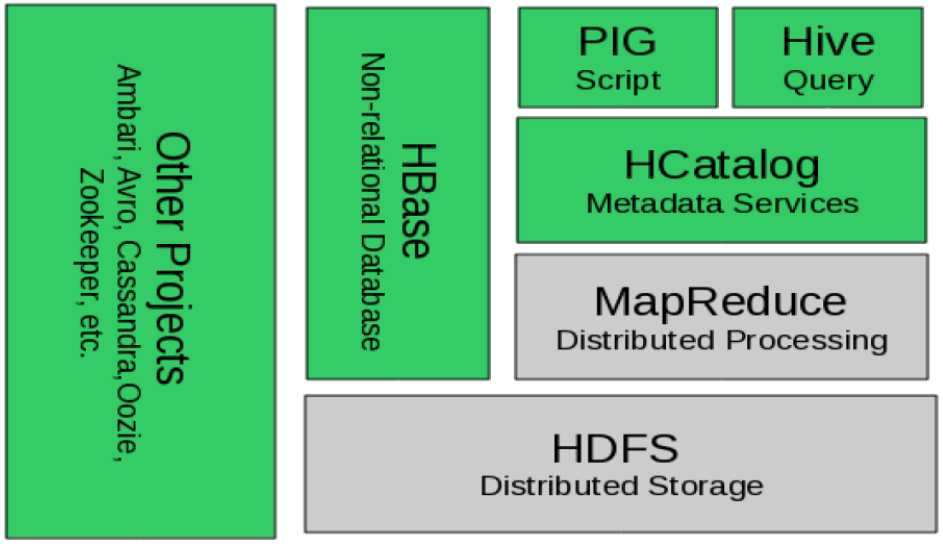


Figure 2.1 The Hadoop 1.0 ecosystem, MapReduce and HDFS are the core components, while other are built around the core.

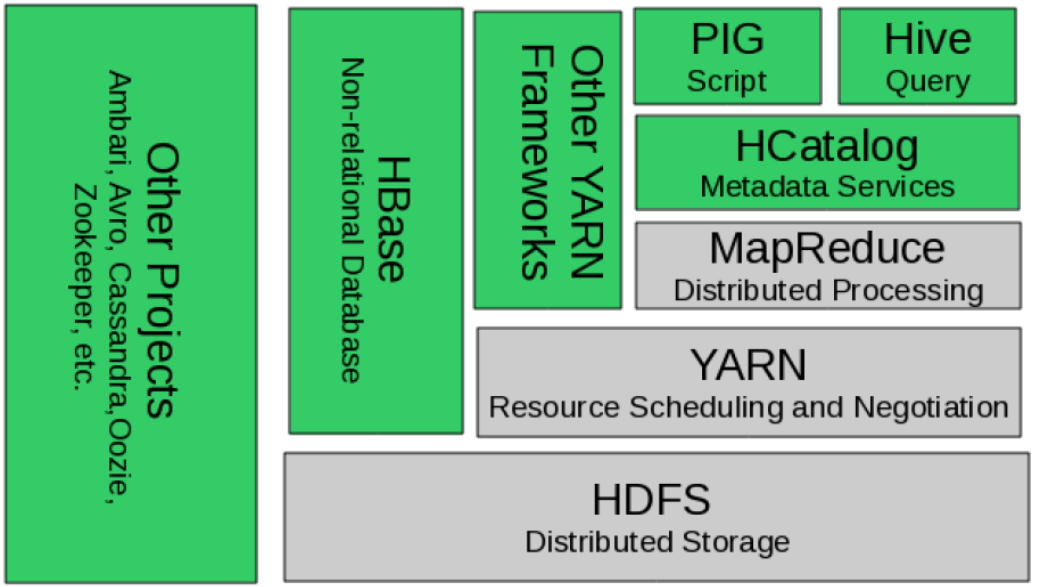


Figure 2.2 YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

Practical Scalability Limits of Hadoop1.0

- Scalability
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
  - Coarse synchronization in Job Tracker
- Single point of failure
  - Failure kills all queued and running jobs
  - Jobs need to be resubmitted by users
- Restart is very tricky due to complex state

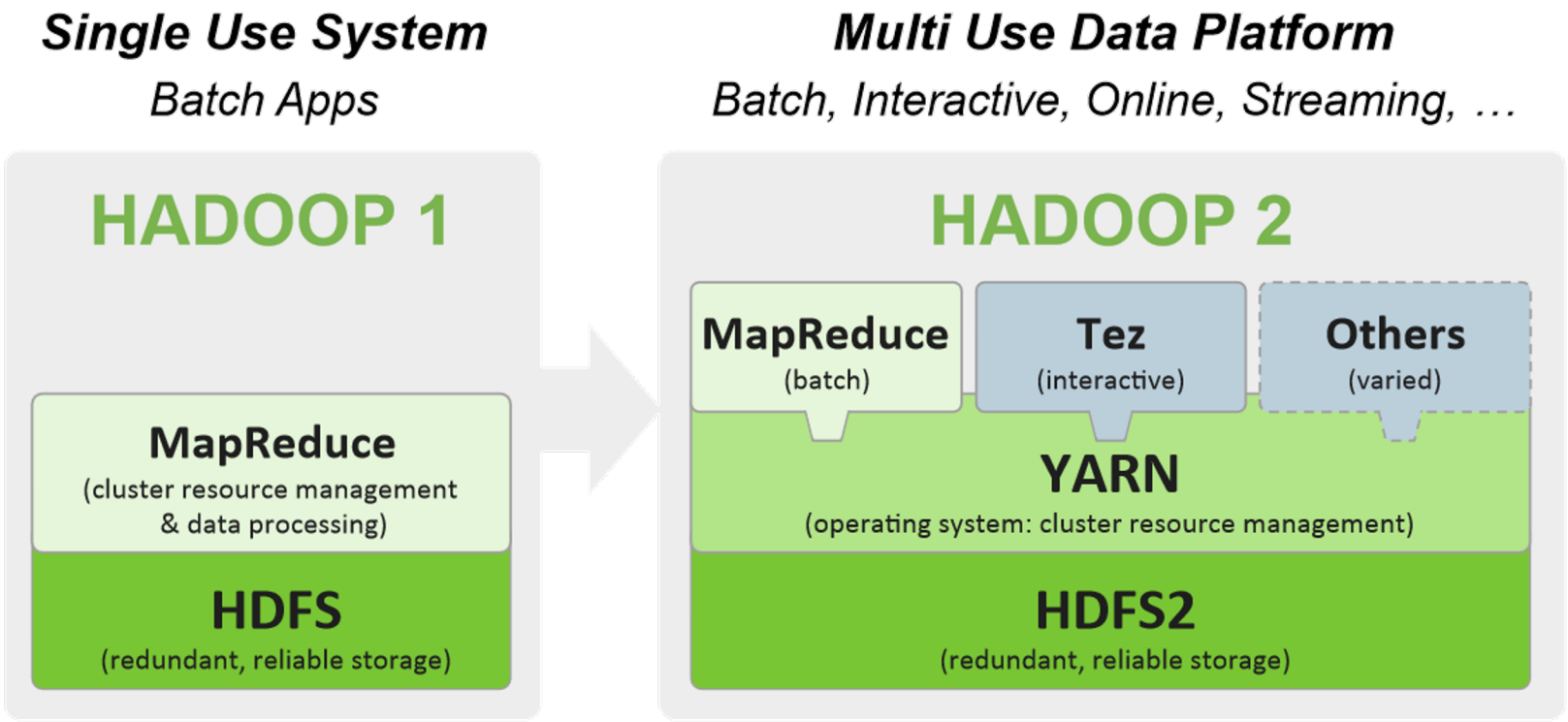
Scalability/Flexibility Issues of the MapReduce/ Hadoop 1.0 Job Scheduling/Tracking

- The MapReduce Master node (or Job-tracker in Hadoop 1.0) is responsible to monitor the progress of ALL tasks of all jobs in the system and launch backup/replacement copies in case of failures
  - For a large cluster with many machines, the number of tasks to be tracked can be huge

  => Master/Job-Tracker node can become the performance bottleneck
- Hadoop 1.0 platform focuses on supporting MapReduce as its only computational model ; may not fit all applications
- Hadoop 2.0 introduces a new resource management/ job-tracking architecture, YARN [1], to address these problems

[1] V.K. Vavilapalli, A.C.Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator,” ACM Symposium on Cloud Computing 2013.

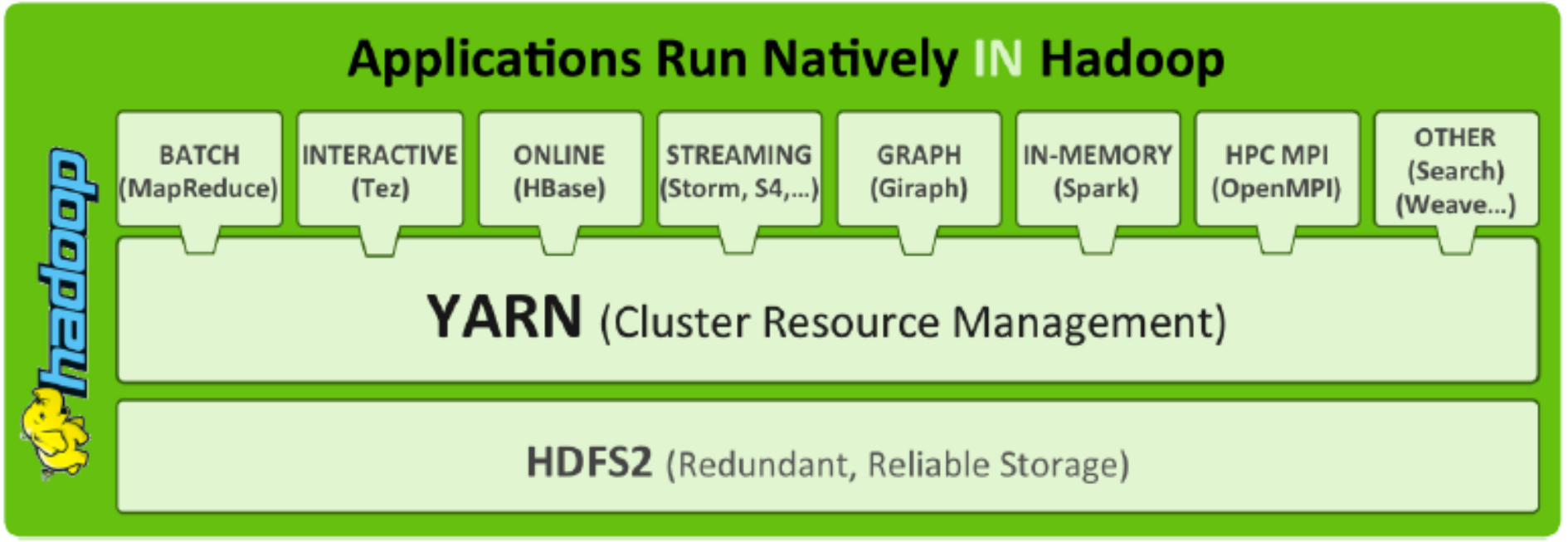
YARN for Hadoop 2.0



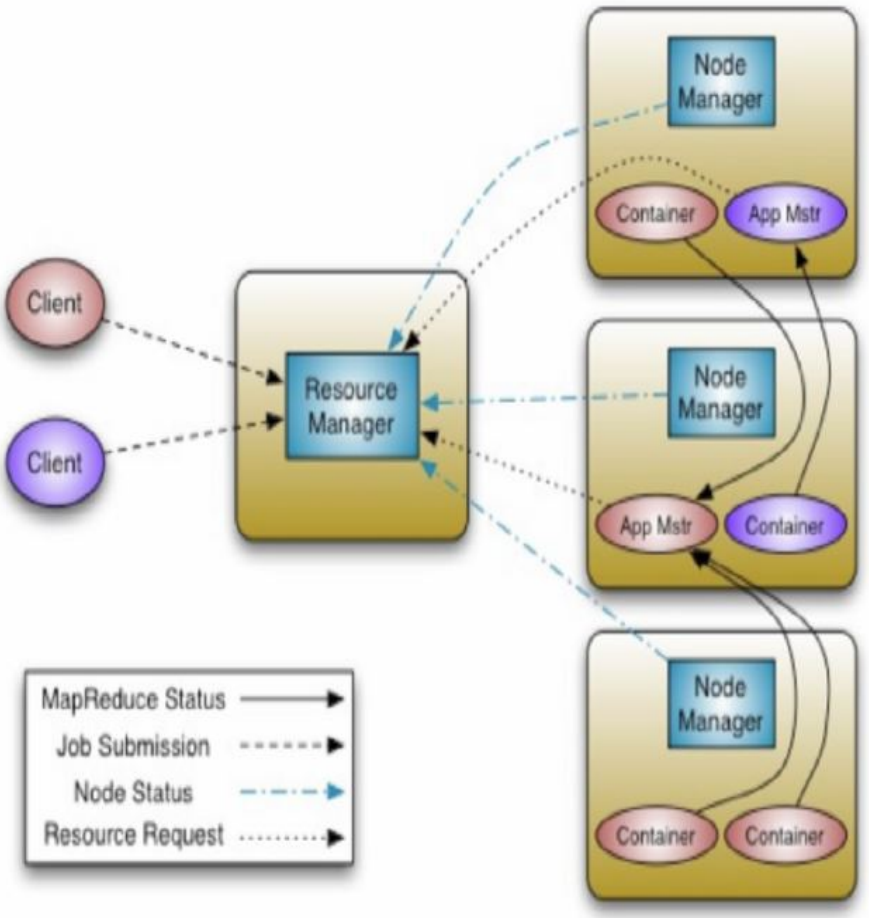
- YARN (Yet Another Resource Negotiator) provides a resource management platform for Cluster to support general Distributed/Parallel Applications/Frameworks beyond the MapReduce computational model.

V. K. Vavilapalli, A. C. Murthy, “Apache Hadoop YARN: Yet Another Resource Negotiator”, in ACM Symposium on Cloud Computing (SoCC) 2013.

A Big Data Processing Stack with YARN



Hadoop2.0/YARN Architectural Overview



- Scalability - Clusters of 6,000-10,000 machines
  - Each machine with 16 cores, 48G/96G RAM, 24TB/36TB disks
  - 100,000+ concurrent tasks
  - 10,000 concurrent jobs

YARN Framework

RescourceManager:

Arbitrates resources among all the applications in the system

NodeManager:

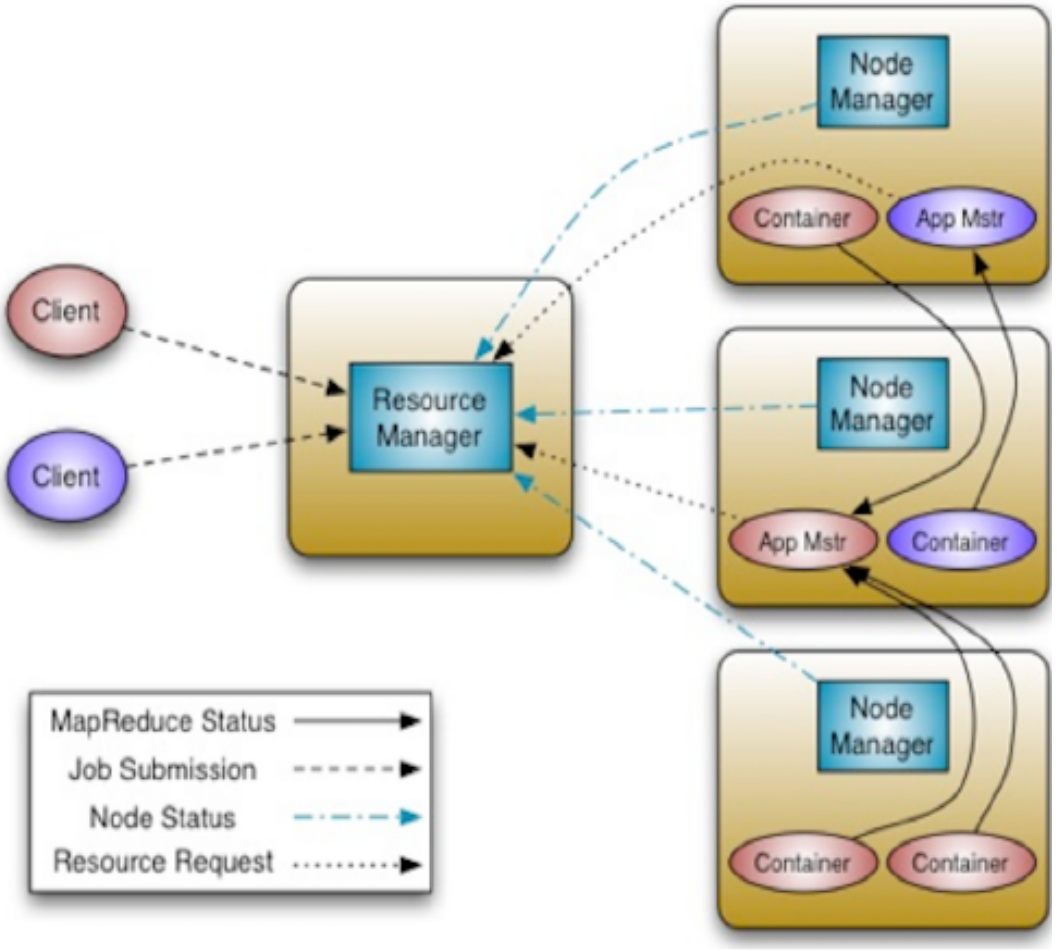
the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage

ApplicationMaster:

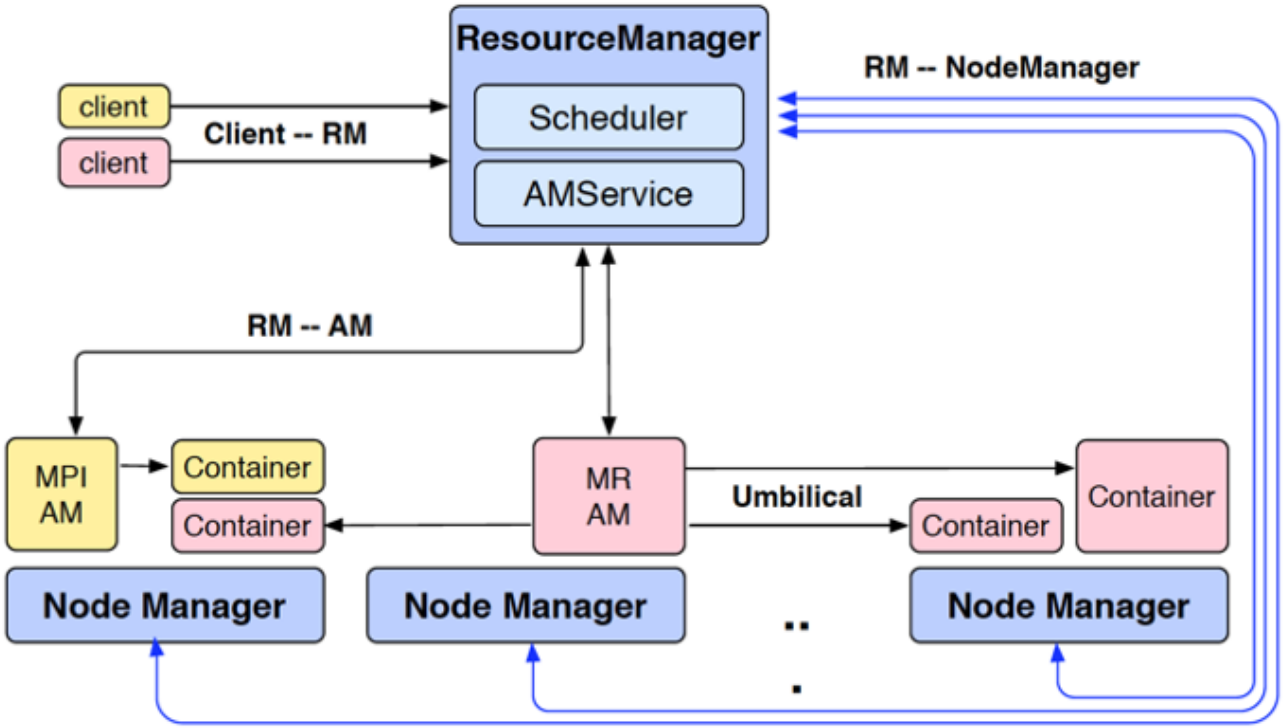
Negatiate appropriate resource containers from the Scheduler, tracking their status and monitoring for progress

Container:

Unit of allocation incorporating resource elements such as memory, cpu, disk, network etc, to execute a specific task of the application (similar to map/reduce slots in MRv1)



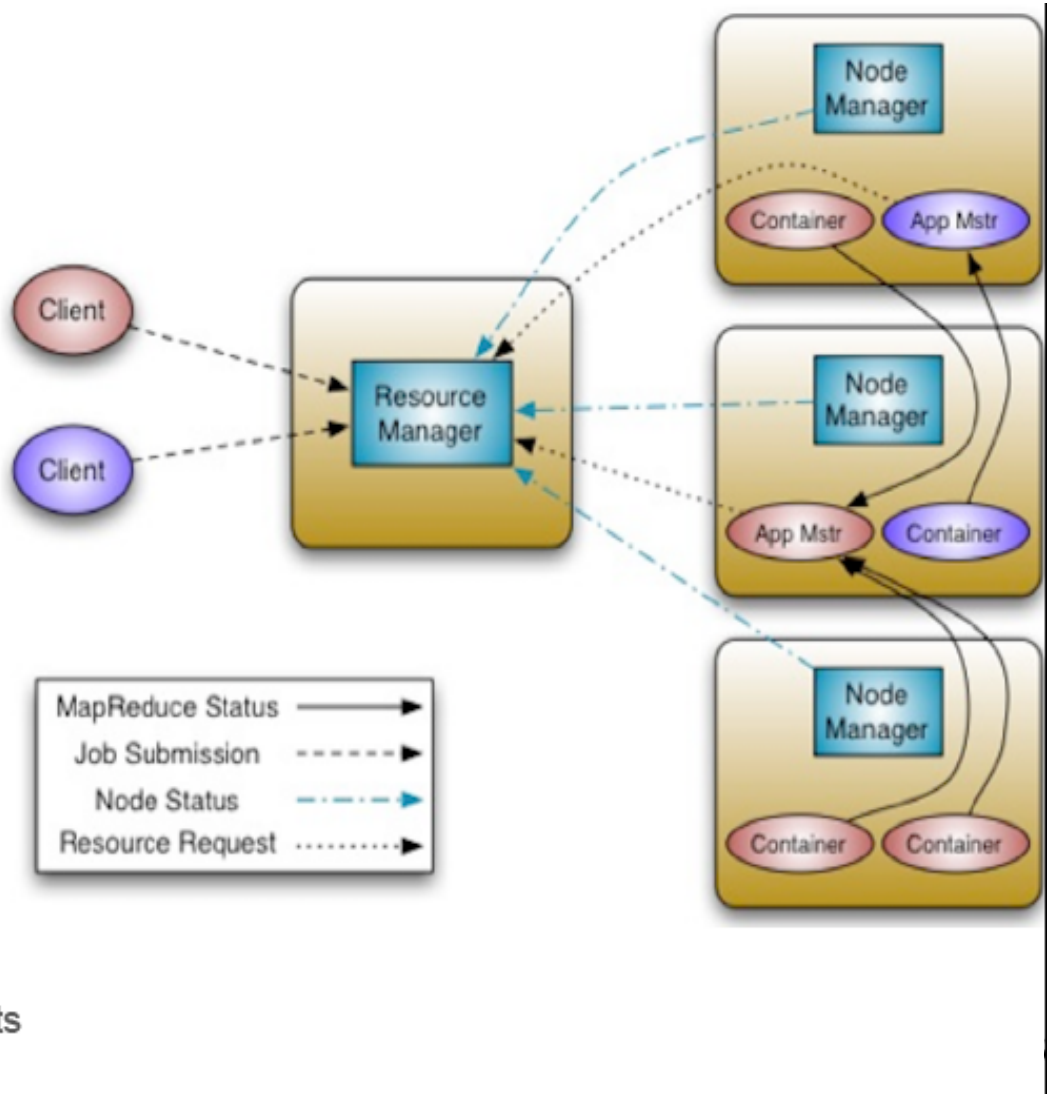
Cluster Resource Management w/ YARN in Hadoop2.0



- Multiple frameworks (Applications) can run on top of YARN to share a Cluster, e.g. MapReduce is one framework (Application), MPI, or Storm are other ones.
- YARN splits the functions of JobTracker into 2 components: resource allocation and job-management (e.g. task-tracking/ recovery):
  - Upon launching, each Application will have its own Application Master (AM), e.g. MR-AM in the figure above is the AM for MapReduce, to track its own tasks and perform failure recovery if needed
  - Each AM will request resources from the YARN Resource Manager (RM) to launch the Application's jobs/tasks (Containers in the figure above) ;
  - The YARN RM determines resource allocation across the entire cluster by communicating with/ controlling the Node Managers (NM), one NM per each machine.

YARN Execution Sequence

1. A client program submits the application
2. ResourceManager allocates a specified container to start the ApplicationMaster
3. ApplicationMaster, on boot-up, registers with ResourceManager
4. ApplicationMaster negotiates with ResourceManager for appropriate resource containers
5. On successful container allocations, ApplicationMaster contacts NodeManager to launch the container
6. Application code is executed within the container, and then ApplicationMaster is responded with the execution status
7. During execution, the client communicates directly with ApplicationMaster or ResourceManager to get status, progress updates etc.
8. Once the application is complete, ApplicationMaster unregisters with ResourceManager and shuts down, allowing its own container process



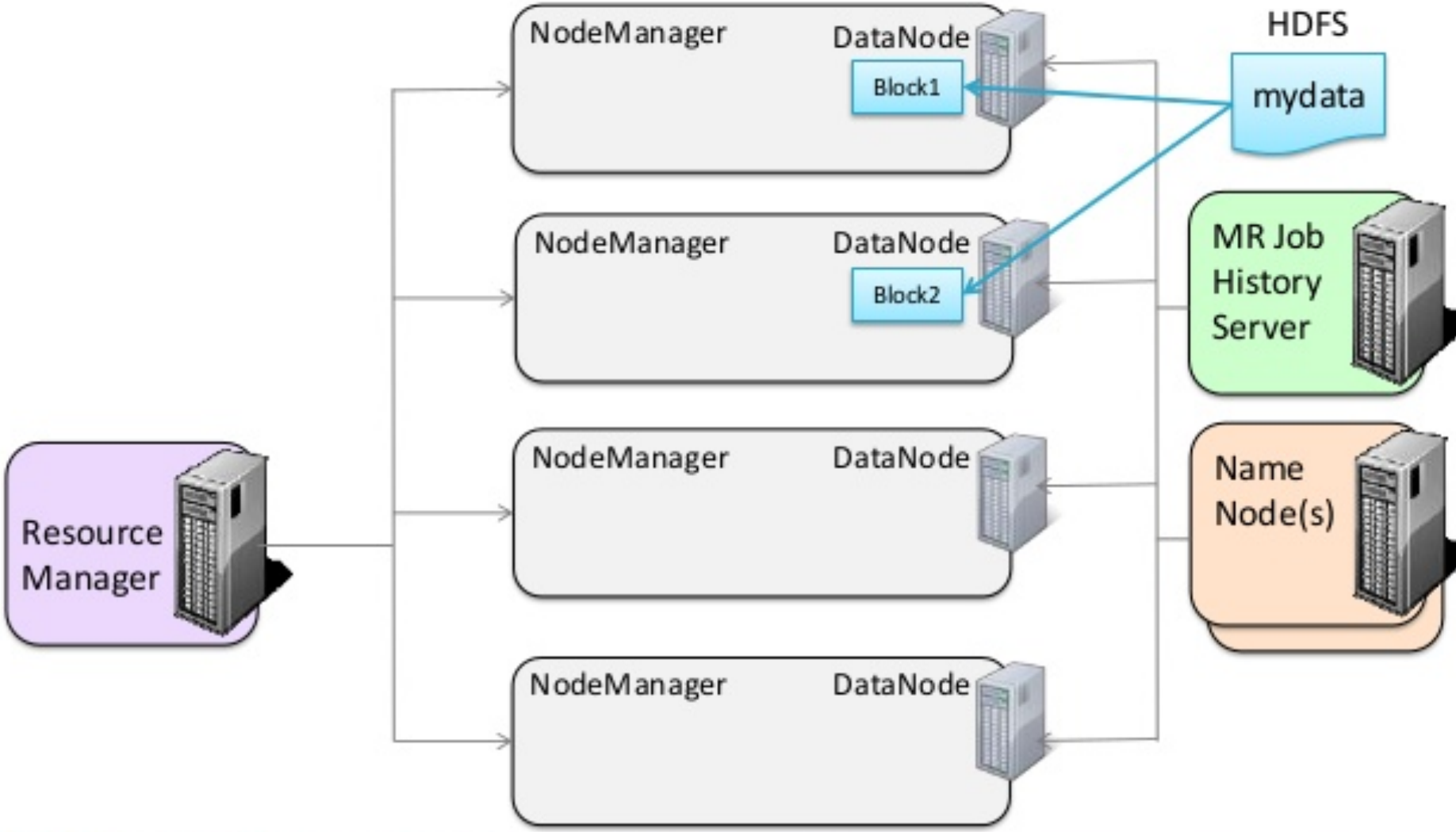
YARN Application Models

- Application Master (AM) per Job
  - Most simple for batch
  - Used by MapReduce (v2)
- Application Master per Session
  - Runs multiple jobs on behalf of the same user
  - Added in Tez ;
  - Also for Spark (one AM per SparkContext, w/ Long-lived enhancement)
- AM as permanent service, supporting Multiple Users
  - Always on, waits around for jobs to come in
  - Used for Impala (with Llama Adapter to support separate-user/queue billing of YARN)

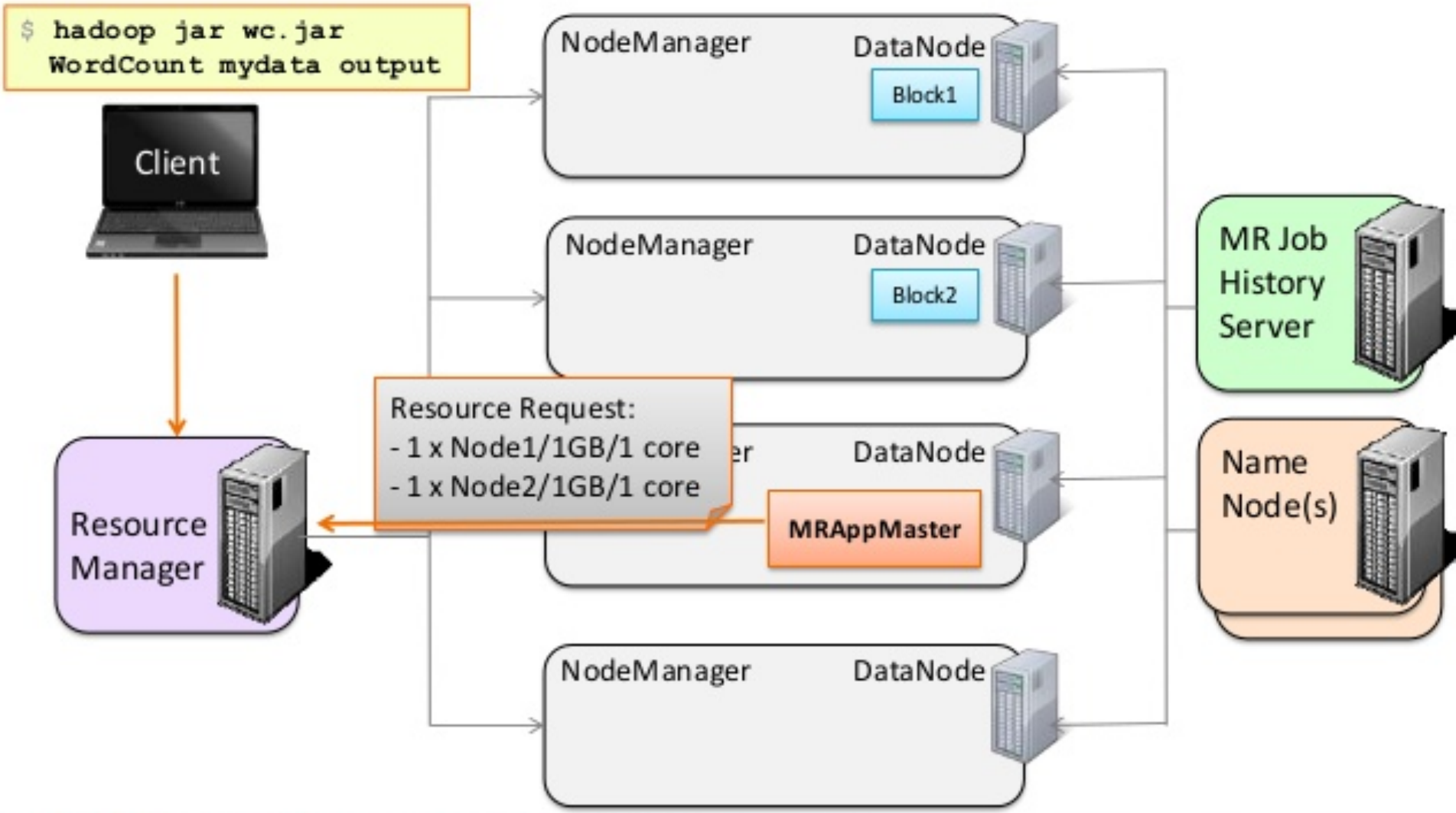
Example: Running MapReduce (v2) on YARN

- Each MapReduce Job has a separate instance of AM
- A Separate MapReduce Job History Server to track MR job history
- YARN runs Shuffle as a persistent, auxiliary service

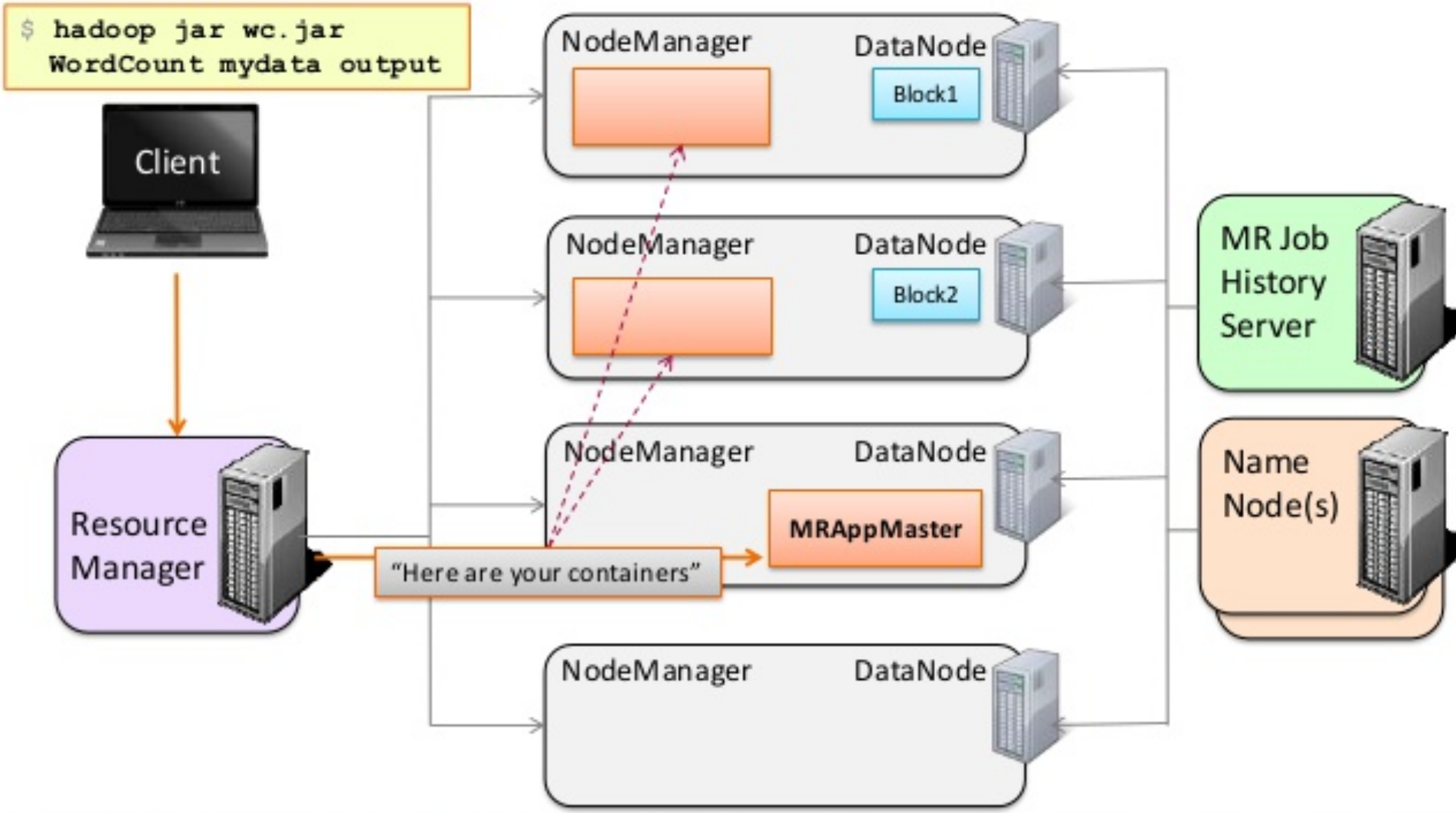
Running a MapReduce Application in MRv2



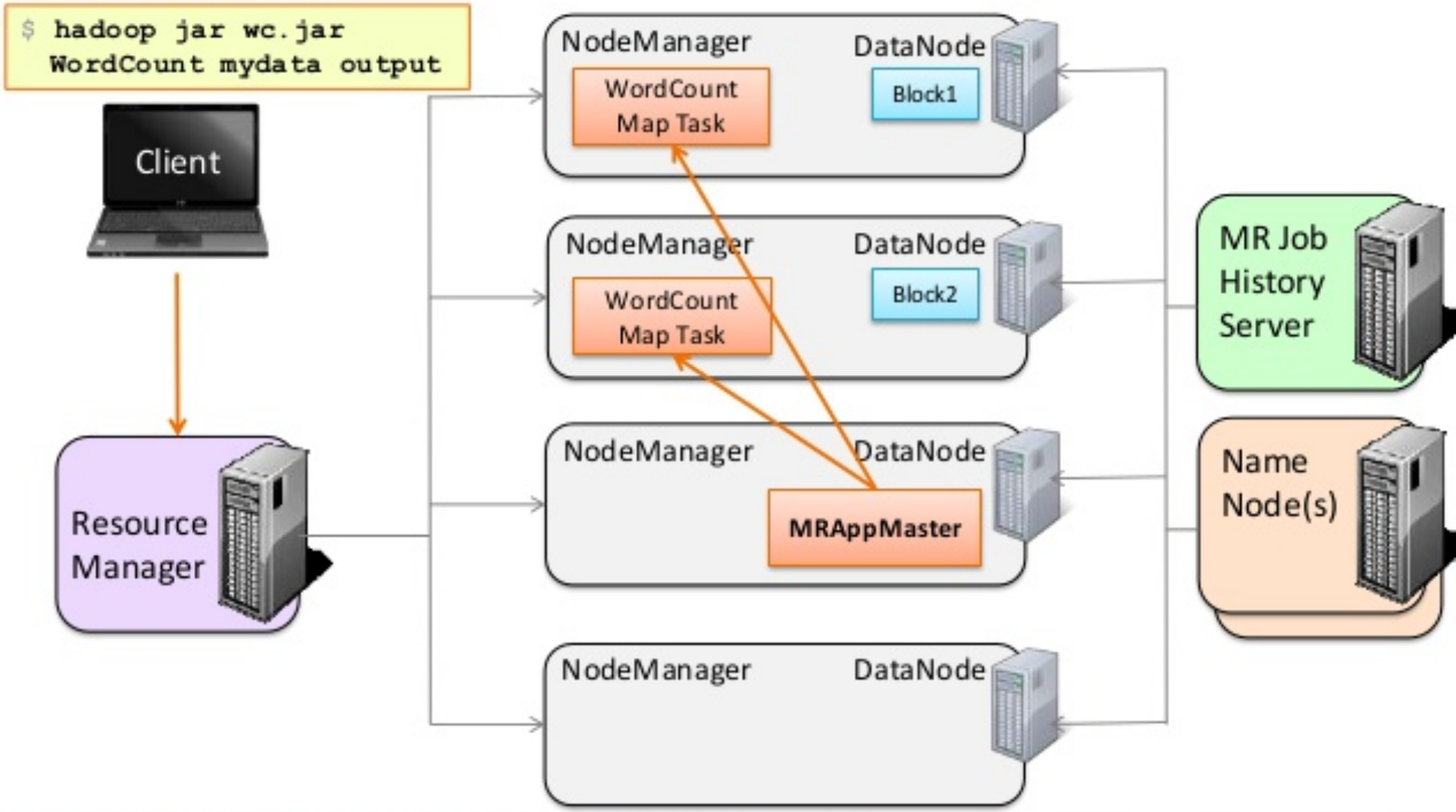
Running a MapReduce Application in MRv2



Running a MapReduce Application in MRv2

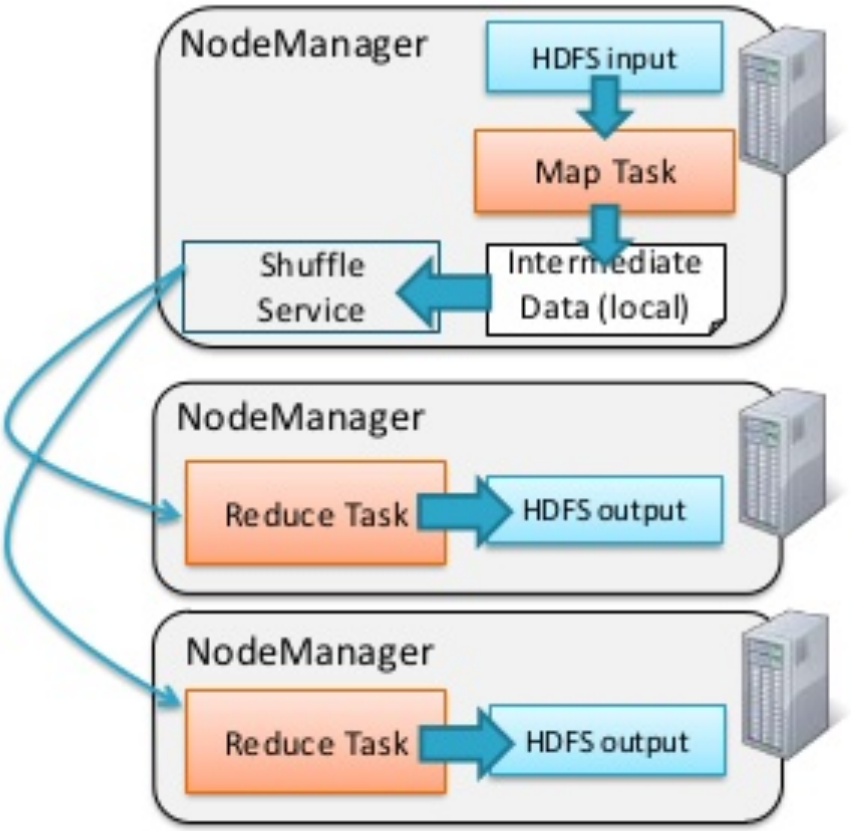


Running a MapReduce Application in MRv2



The MapReduce Framework on YARN

- **In YARN, Shuffle is run as an auxiliary service**
  - Runs in the NodeManager JVM as a persistent service

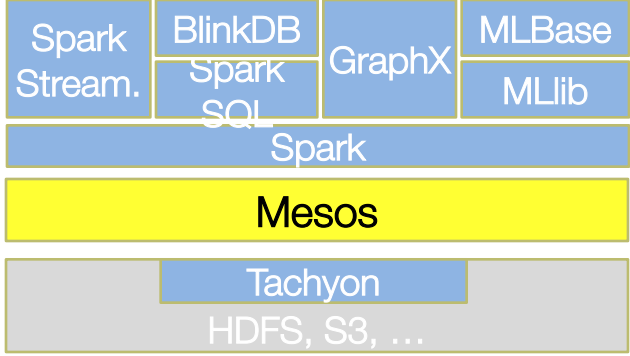


Hadoop 2.0 vs. Hadoop1.0

- Hadoop 2.0 includes YARN's Multi-tenant Support for different Big Data Processing Frameworks
- YARN Fault Tolerance and Availability
  - Resource Manager
    - No single point of failure – state saved in ZooKeeper
    - Application Masters are restarted automatically on RM restart
  - Application Master
    - Optional failover via application-specific checkpoint
    - MapReduce applications pick up where they left off via state saved in HDFS
- Wire Compatibility
  - Protocols are wire-compatible
  - Old clients can talk to new servers
  - Rolling upgrades
- Besides YARN, Hadoop 2.0 also supports High Availability and Federation
  - High Availability takes away the Single Point of failure from HDFS Namenode and introduces the concept of the QuorumJournalNodes to sync edit logs between active and standby Namenodes
  - Federation allows multiple independent namespaces (private namespaces, or Hadoop as a service)

Apache Mesos

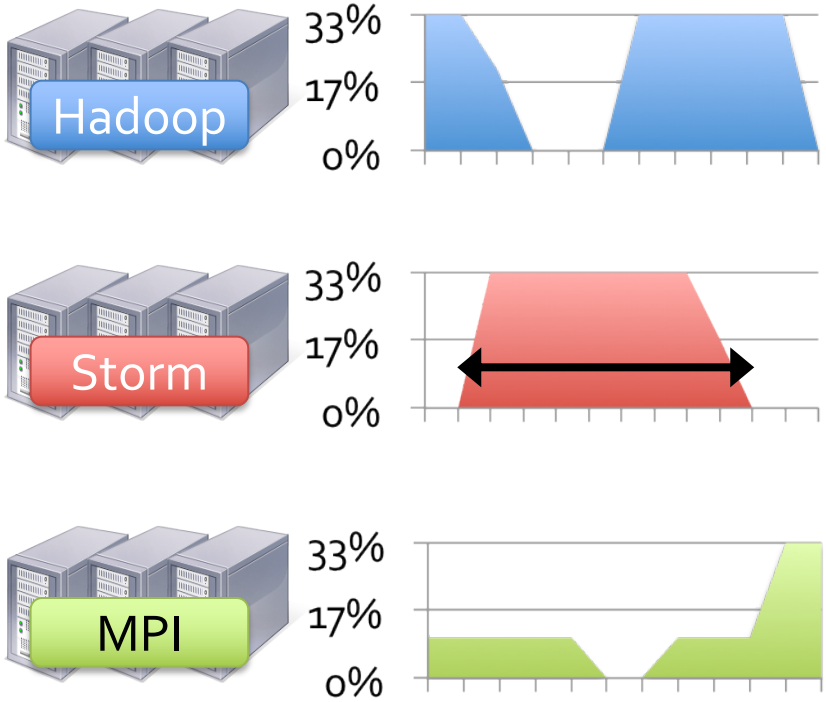
(http://mesos.apache.org)



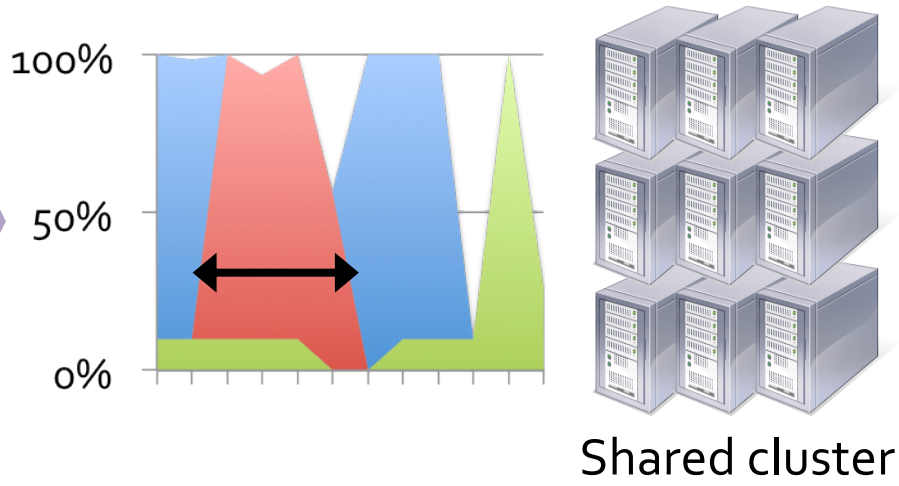
- Another competing Cluster Resource Management platform
- Enable multiple frameworks to share same cluster resources (e.g., MapReduce, Storm, Spark, HBase, etc)
- Originated from UCBerkeley's BDAS project ;
  - B. Hindman et al, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center", Usenix NSDI 2011.
- Hardened via Twitter's large scale in-house deployment
  - 6,000+ servers,
  - 500+ engineers running jobs on Mesos
- Third party Mesos schedulers
  - AirBnB's Chronos ; Twitter's Aurora
- Mesospehere: startup to commercialize Mesos

Motivation of Mesos

Previously: Static partitioning of a cluster among different big data processing frameworks



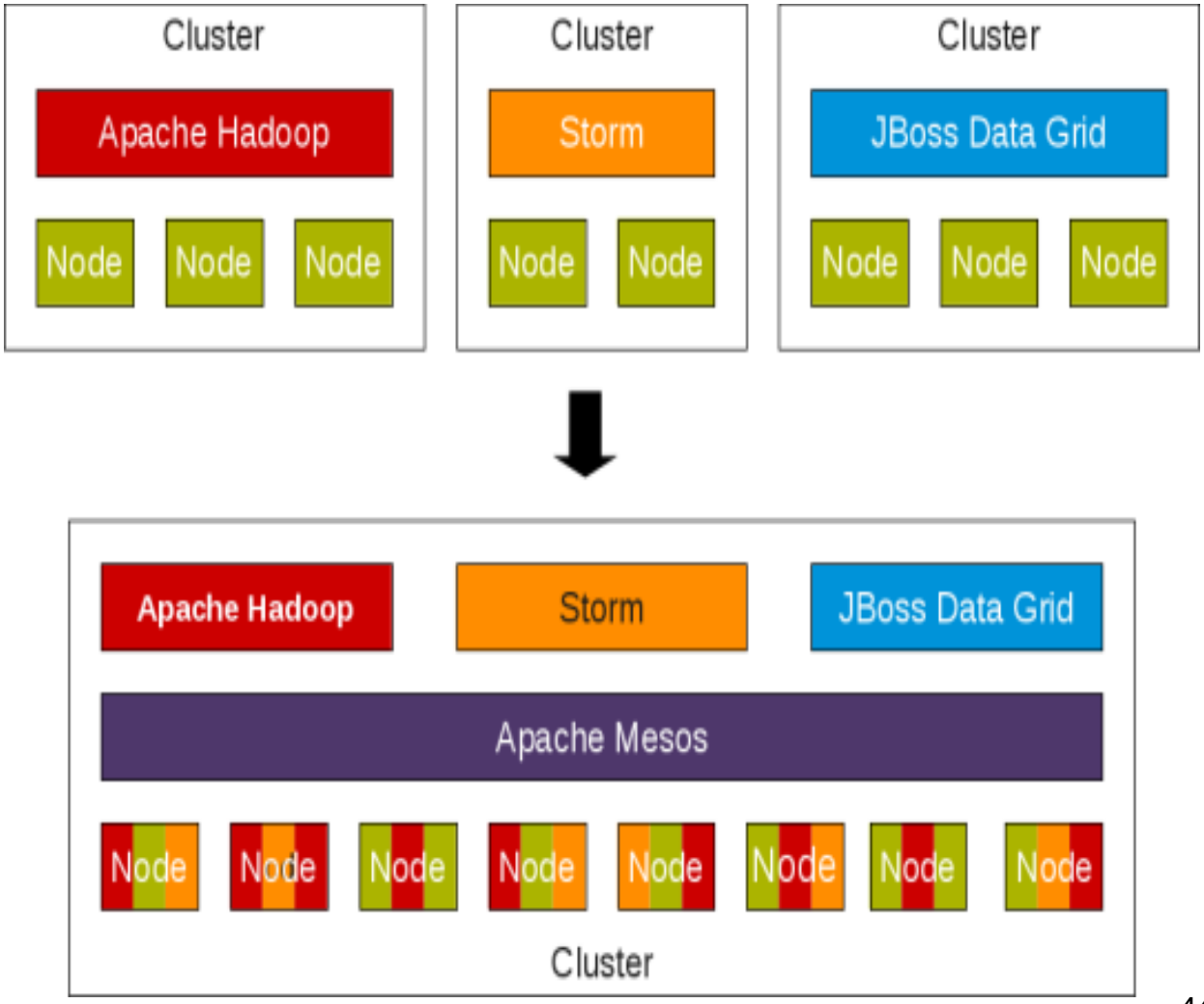
Mesos aims to achieve dynamic sharing of cluster across different frameworks



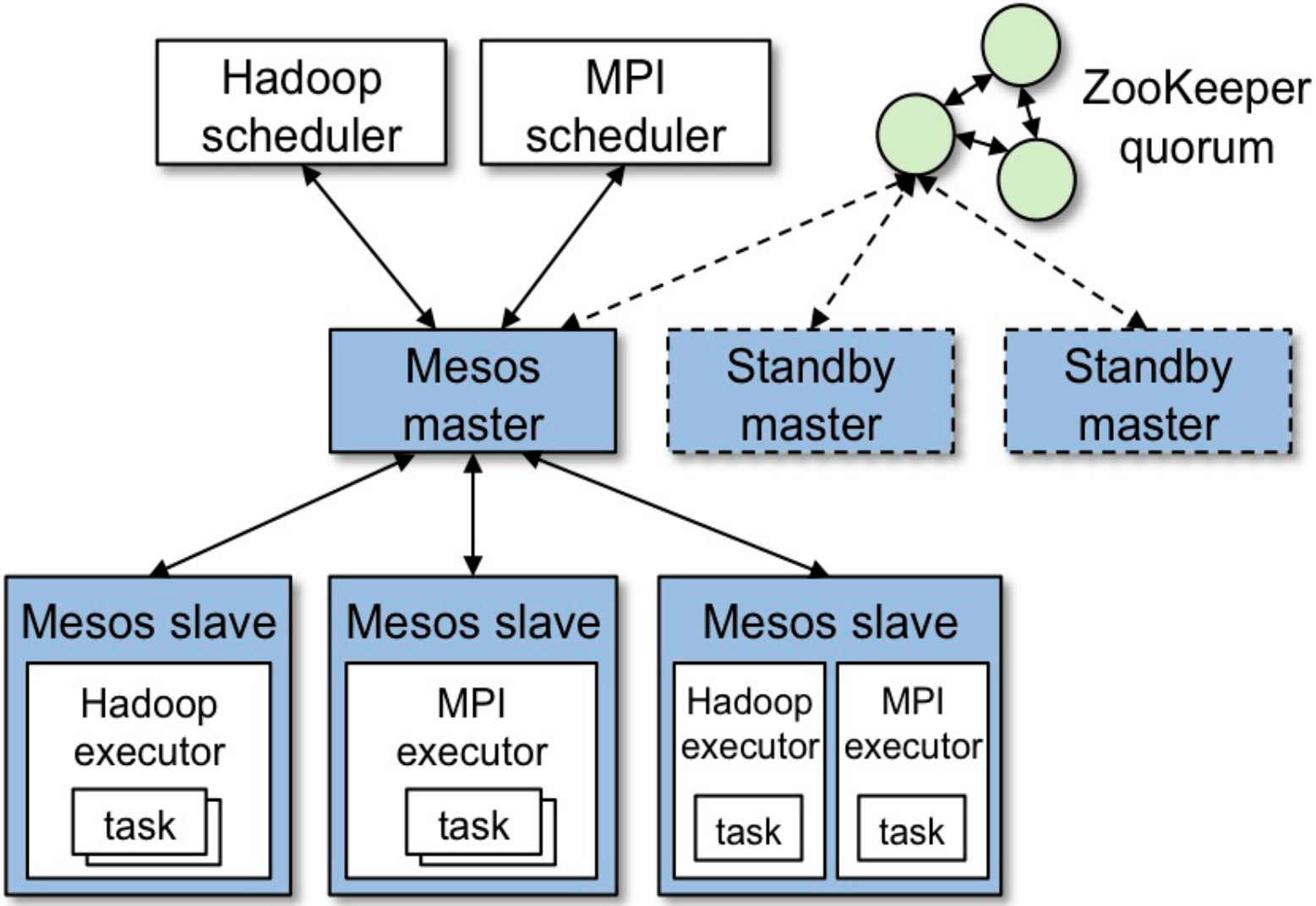
- Hard to fully utilize machines (e.g., X GB RAM & Y CPUs)
- Hard to scale elastically (to take advantage of statistical multiplexing)
- Hard to deal with failures

Mesos as a Data-Center "Kernel"

- Like YARN, Mesos provides a Node Abstraction of the entire Cluster
- Like YARN, Mesos is a common resource sharing layer over which diverse frameworks can run



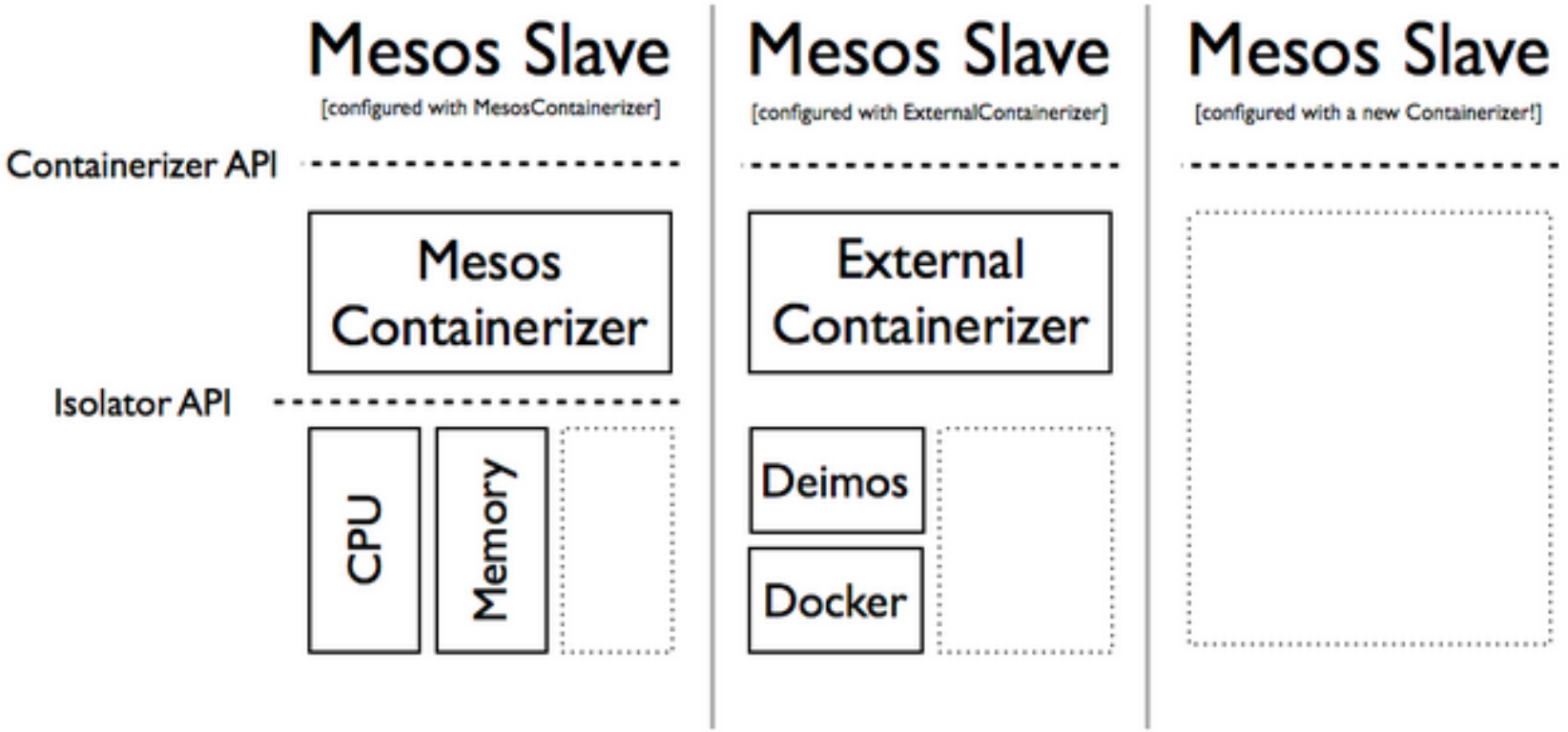
System Architecture of Mesos



Framework Isolation

- Mesos uses OS isolation mechanisms, such as Linux containers and Solaris projects
- Containers currently support CPU, memory, IO and network bandwidth isolation
- Not perfect, but much better than no isolation

Mesos' use of Container Technology

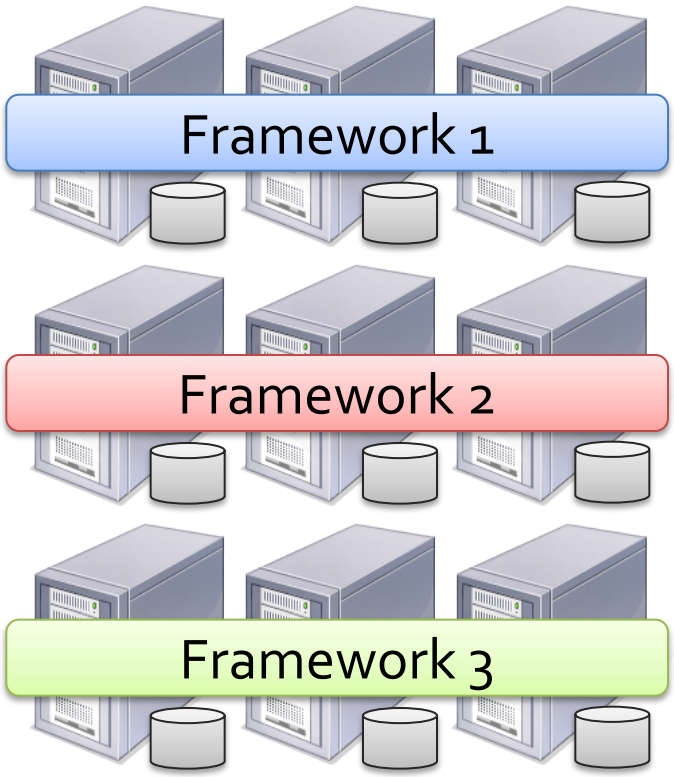


Design Elements

- Fine-grained sharing:
  - Allocation at the level of *tasks* within a job
  - Improves utilization, latency, and data locality
- Resource offers:
  - Simple, scalable application-controlled scheduling mechanism

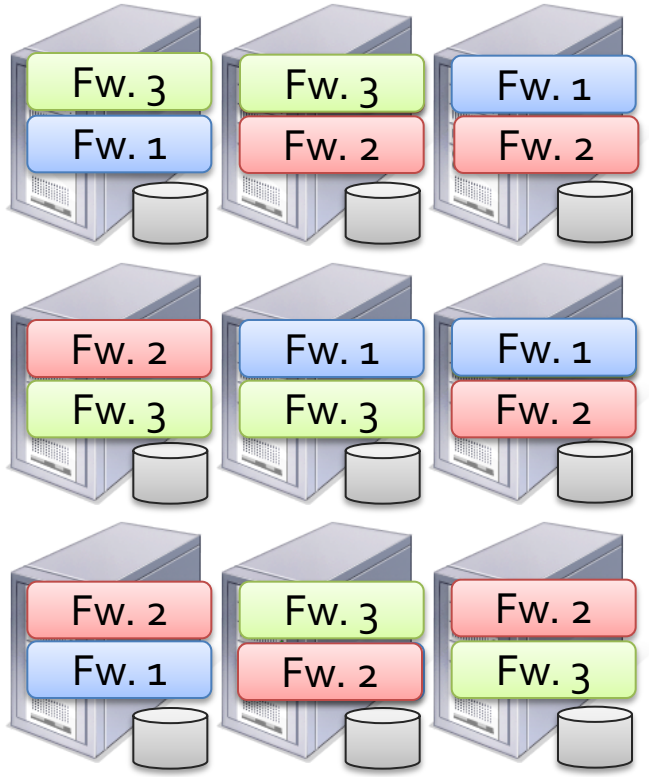
Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Storage System (e.g. HDFS)

Fine-Grained Sharing (Mesos):



Storage System (e.g. HDFS)

+ Improved utilization, responsiveness, data locality

Element 2: Resource Offers

- Option: Global scheduler
  - Frameworks express needs in a specification language, global scheduler matches them to resources

+ Can make optimal decisions

– Complex: language must support all framework needs

– Difficult to scale and to make robust

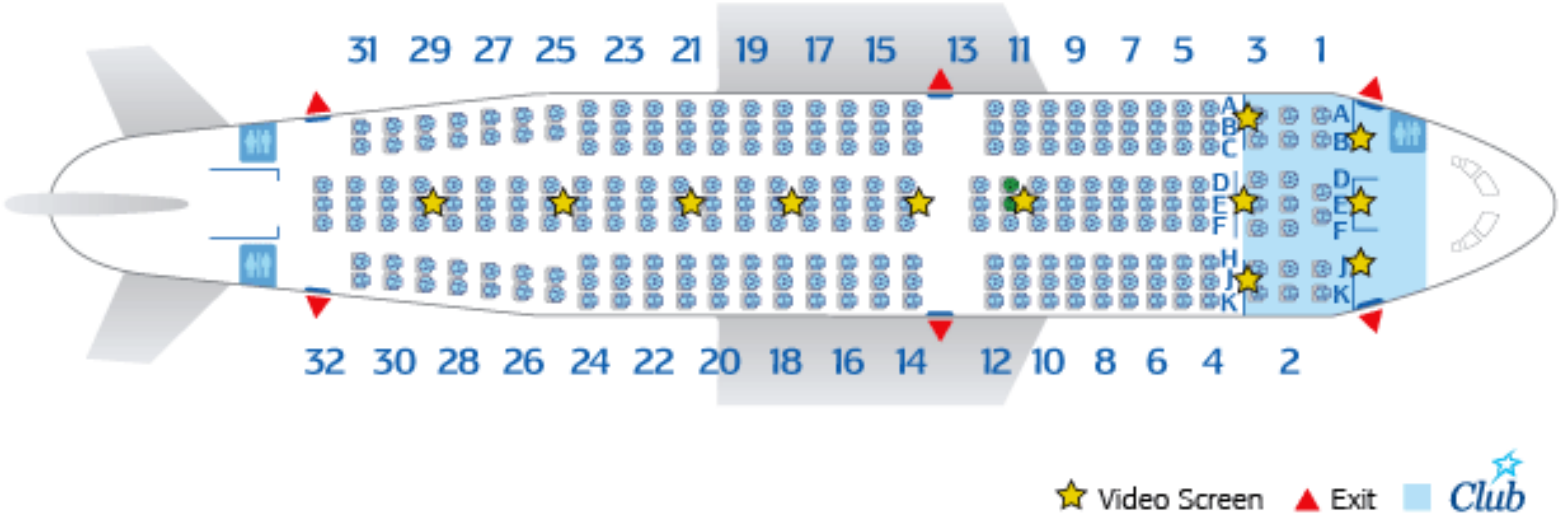
– Future frameworks may have unanticipated needs

Element 2: Resource Offers

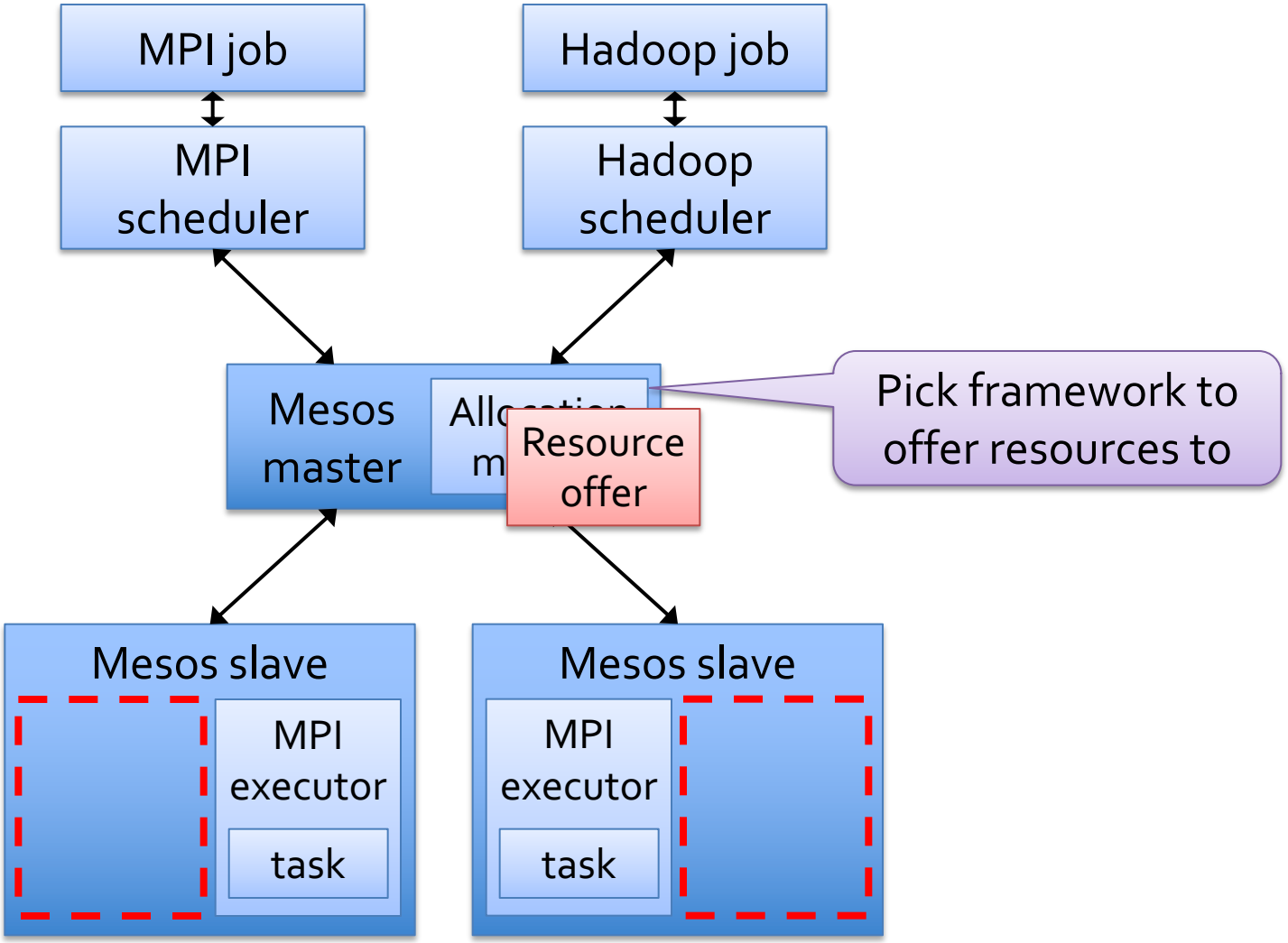
- Mesos: Resource offers
  - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch

  \+ Keep Mesos simple, let it support future frameworks

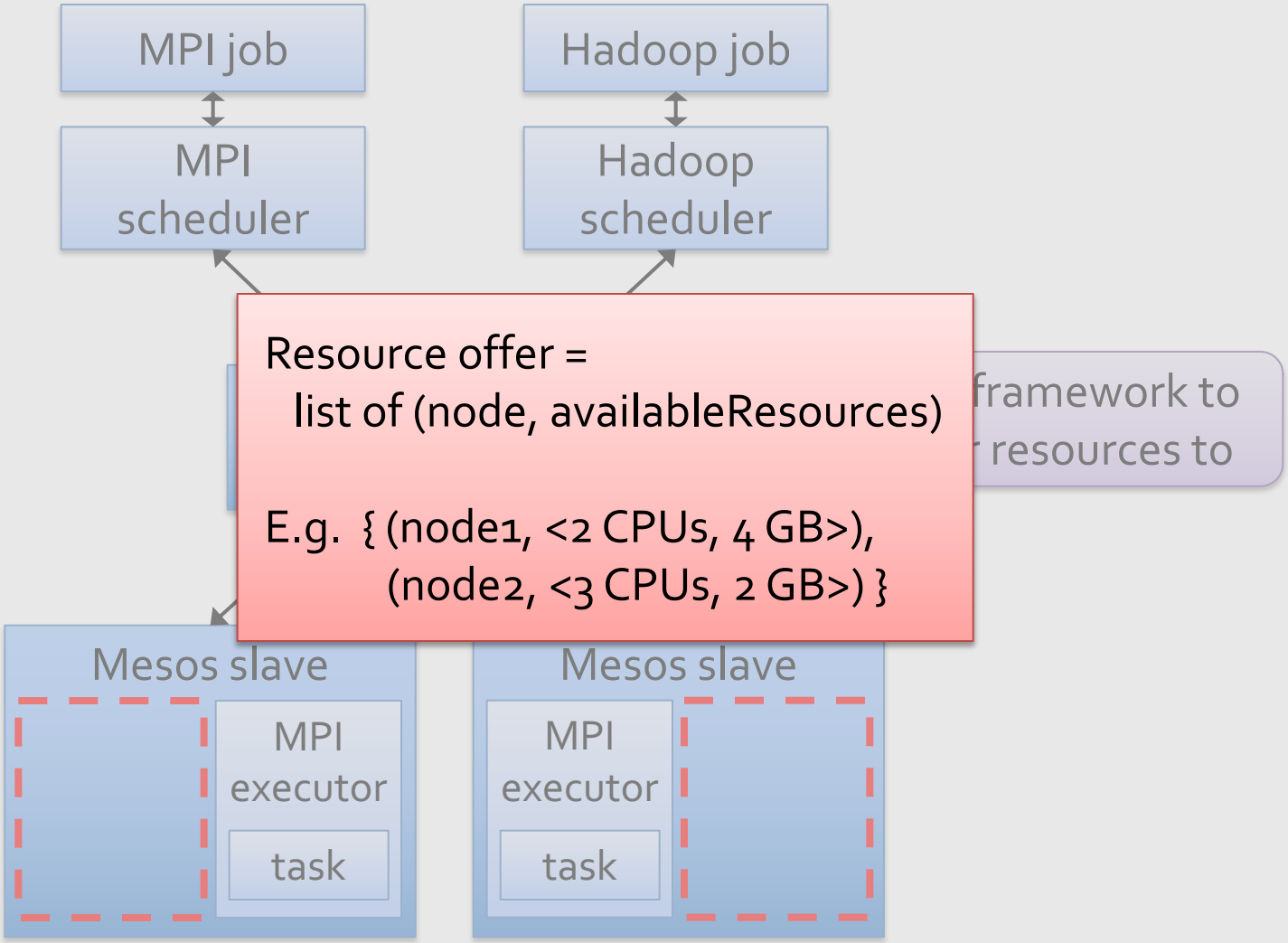
  \- Decentralized decisions might not be optimal



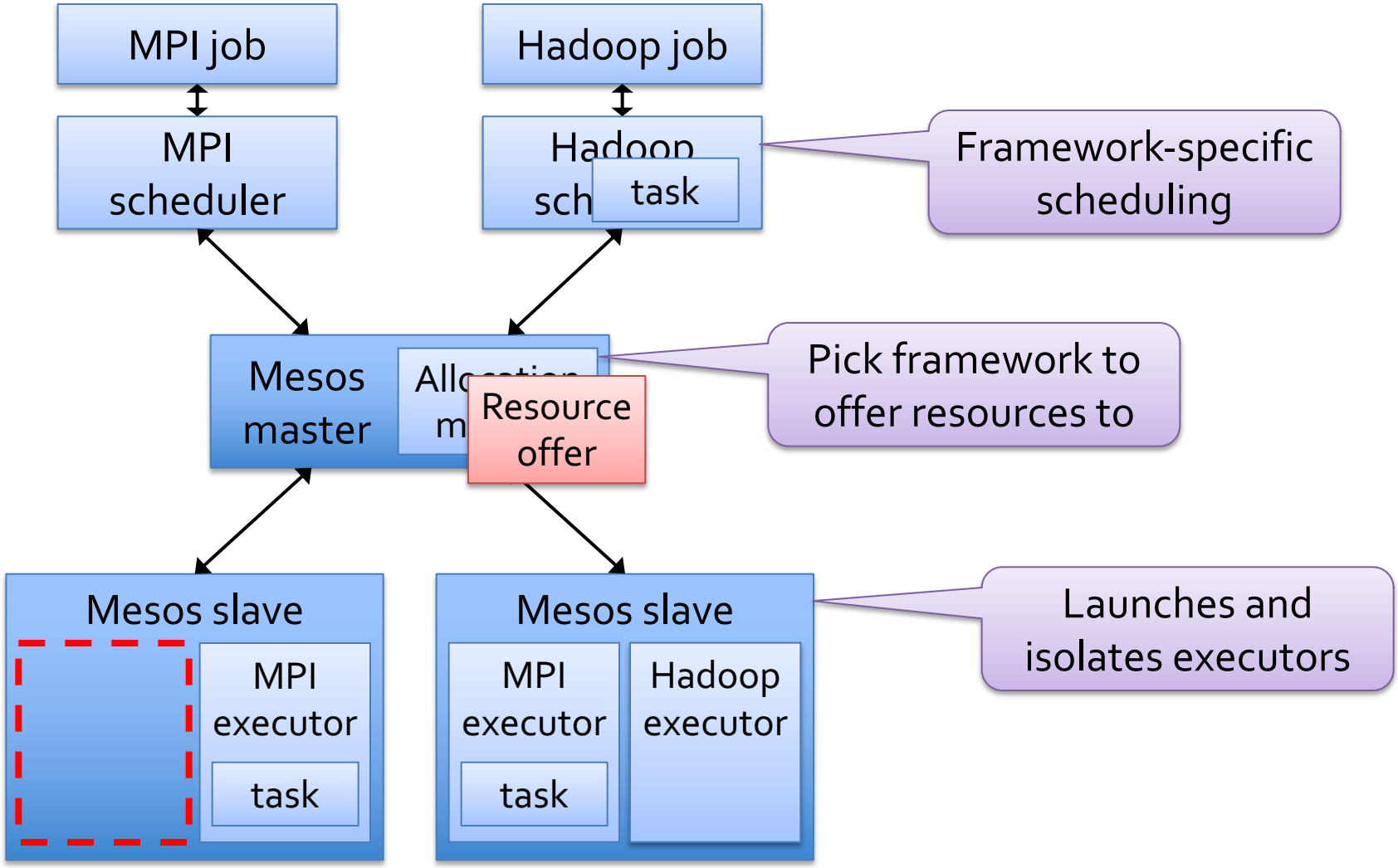
Mesos Architecture



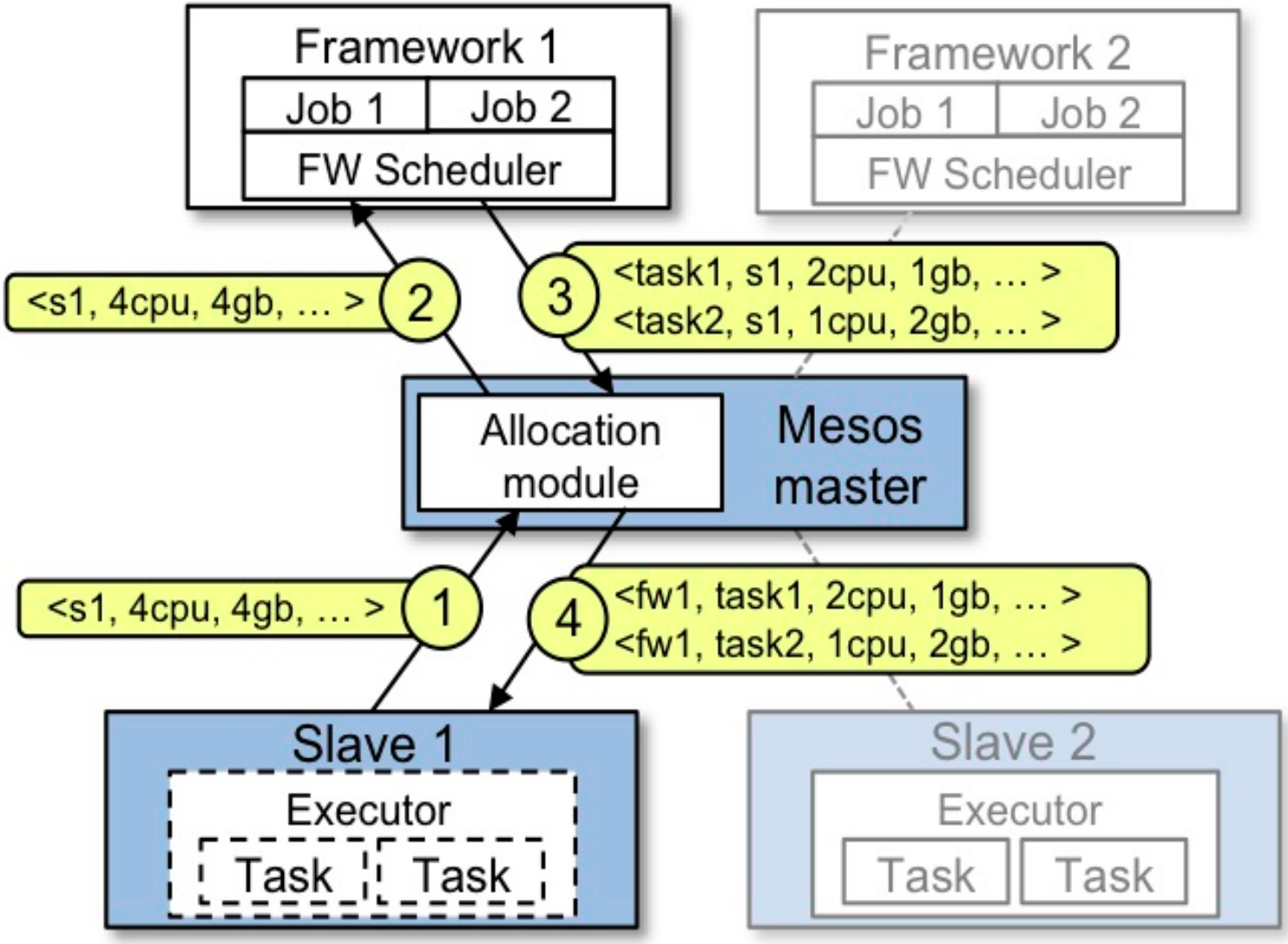
Mesos Architecture



Mesos Architecture



Another Resource Offering Example



Optimization: Filters

- Let frameworks short-circuit rejection by providing a predicate on resources to be offered
  - » E.g. "nodes from list L" or "nodes with > 8 GB RAM"
  - » Could generalize to other hints as well
- Ability to reject still ensures *correctness* when needs cannot be expressed using filters

Revocation

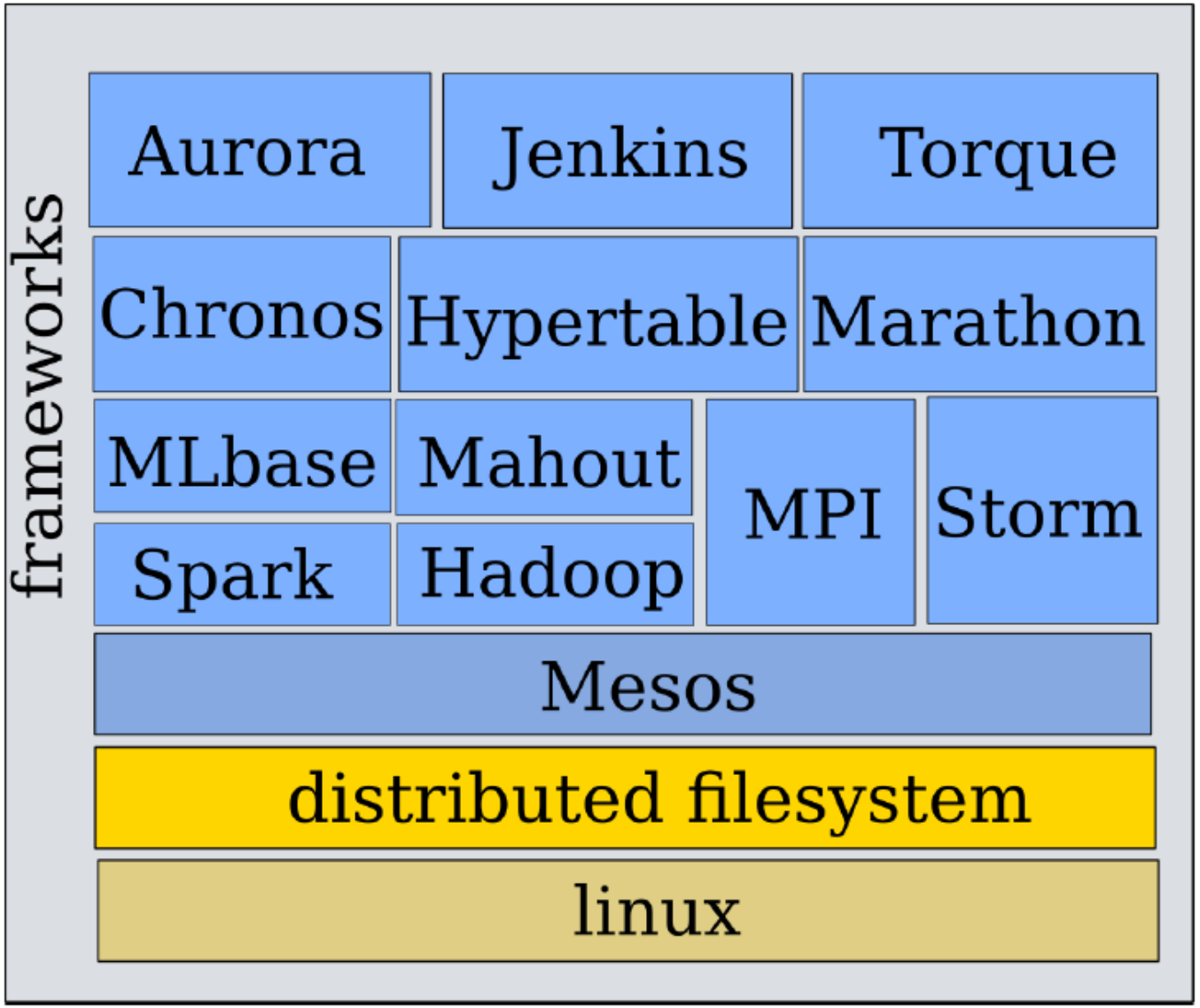
- Mesos allocation modules can revoke (kill) tasks to meet organizational SLOs
- Framework given a grace period to clean up
- "Guaranteed share" API lets frameworks avoid revocation by staying below a certain share

Mesos API

Scheduler Callbacks	Scheduler Actions
resourceOffer(offerId, offers) offerRescinded(offerId) statusUpdate(taskId, status) slaveLost(slaveId)	replyToOffer(offerId, tasks) setNeedsOffers(bool) setFilters(filters) getGuaranteedShare() killTask(taskId)

Executor Callbacks	Executor Actions
launchTask(taskDescriptor) killTask(taskId)	sendStatus(taskId, status)

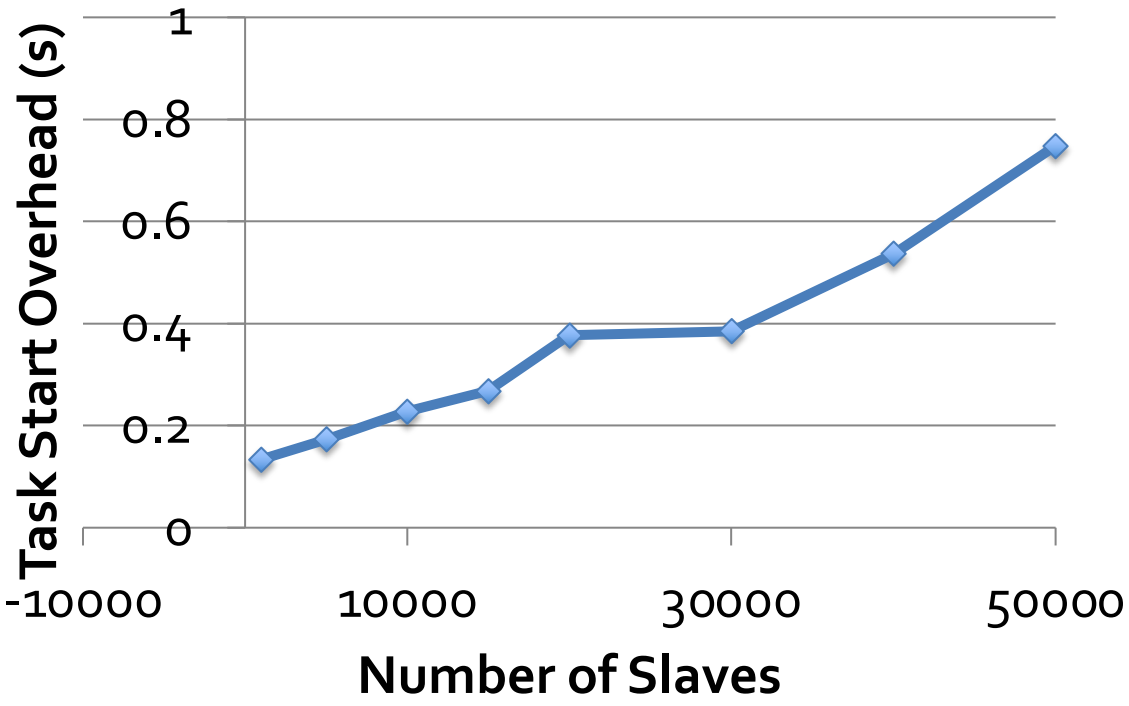
A Big Data Processing Stack w/ Mesos



Scalability

Mesos only performs *inter-framework* scheduling (e.g. fair sharing), which is easier than intra-framework scheduling

Result:
Scaled to 50,000 emulated slaves, 200 frameworks, 100K tasks (30s len)



Fault Tolerance

- Mesos master has only *soft state*: list of currently running frameworks and tasks
- Rebuild when frameworks and slaves re-register with new master after a failure

Result: fault detection and recovery in ~10 sec

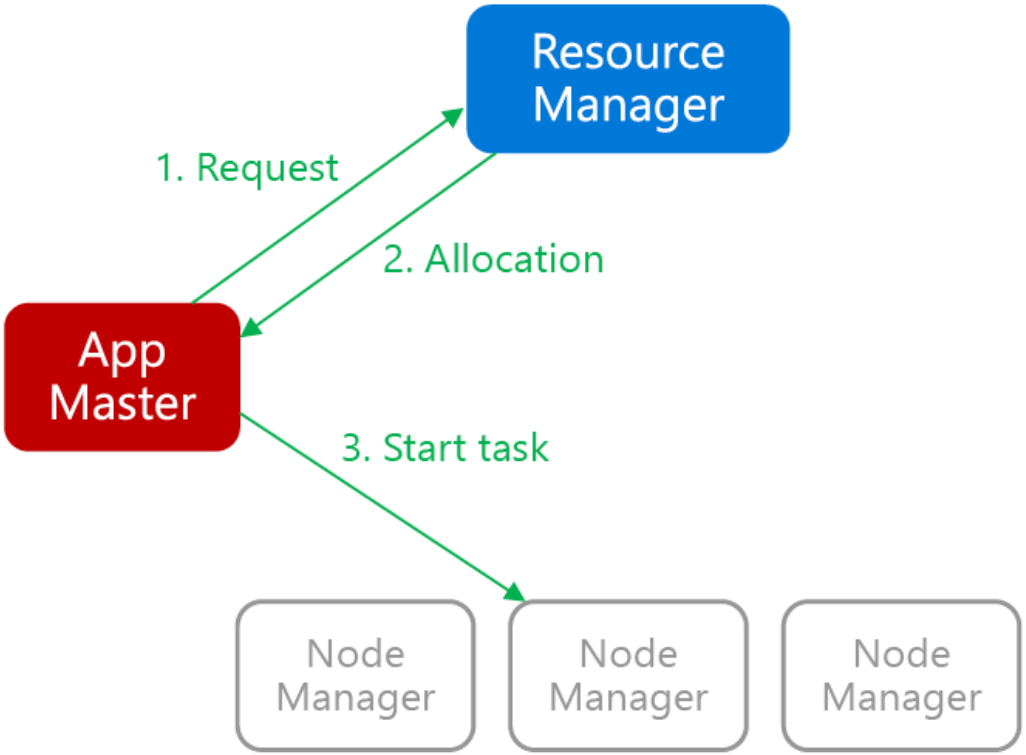
Mesos Implementation Statistics

- 20,000 lines of C++
- Master failover using ZooKeeper
- Frameworks ported: Hadoop1.0, MPI, Storm, etc
- Specialized framework: Spark, for iterative jobs (up to 20 × faster than Hadoop)
- Open source under Apache license

Other Schedulers/ Resource Management Platforms for Big Data Processing Clusters

Approach 1: Centralized Resource Management

[YARN, Mesos, Omega, Borg]



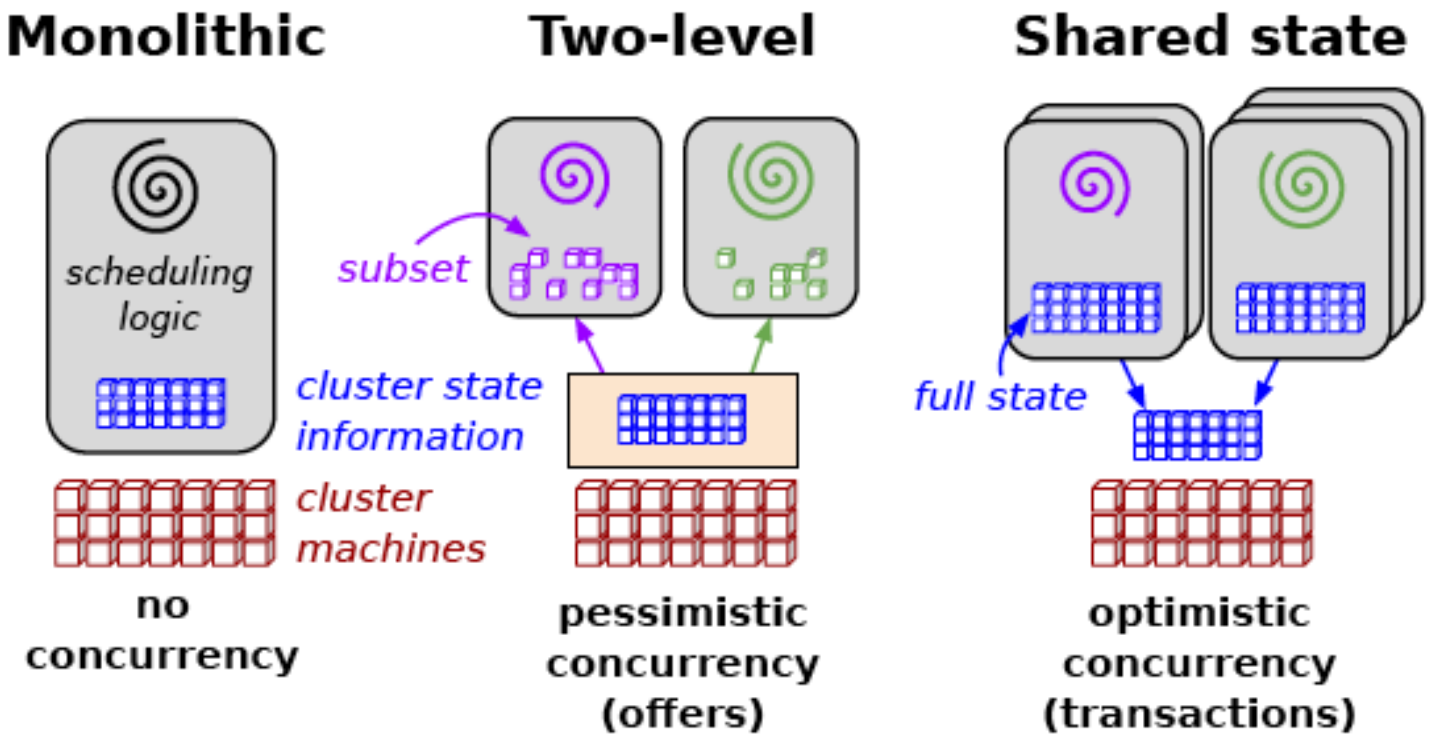
- All scheduling decisions go through the central RM
- The RM resolves all conflicts and guarantees resources to applications

M. Schwarzkopf, A. Konwinski, M.Abd-El-Malek, J. Wilkes, “Omega: flexible, scalable schedulers for large compute clusters,” Eurosys 2013

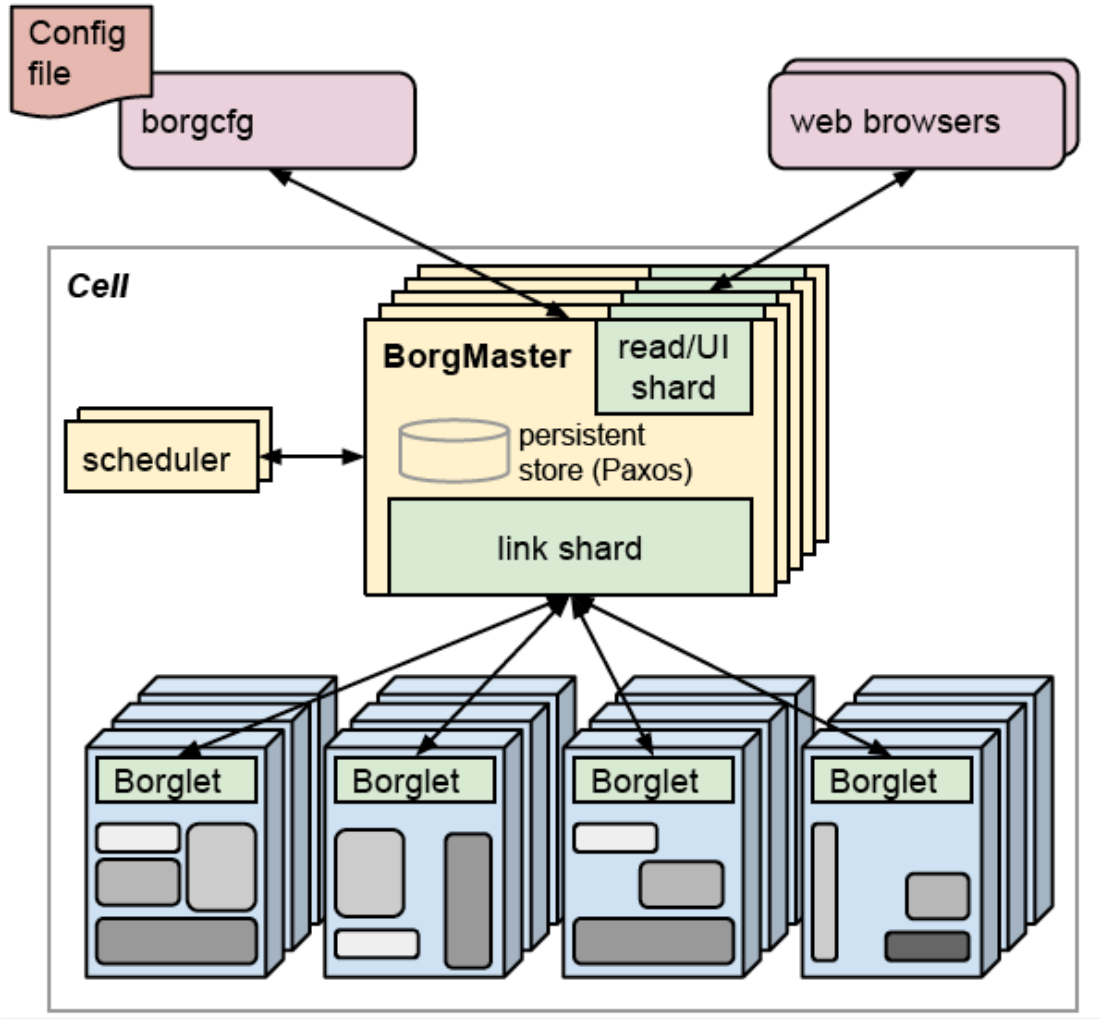
A. Verma, L. Pedrosa, “Large-scale cluster management at Google with Borg”, Eurosys 2015

Design Options for Centralized Resource Management:

Monolithic[Hadoop1.0, YARN] vs.Two-level[Mesos] vs.Shared-state[Omega, Borg



High-level Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Borg Architecture - Borgmaster

- Each cell contains a Borgmaster
- Each Borgmaster consists of 2 processes:
  - Main Borgmaster process
  - Scheduler
- Multiple replicas of each Borgmaster
- Role of (elected leader) Borgmaster:
  - submission of job, termination of any of job's task

Borg Architecture - Borglet

- Local Borg agent on every cell
  - starts/stops/restarts tasks
  - Manages local resources
  - Rolls over debug logs
- Polled by Borgmaster to get machine's current state
- If a Borglet does not respond to several poll messages, it is marked as down
  - Tasks re-distributed
  - If communication is restored, Borgmaster tells Borglet to kill rescheduled tasks

How does Borg work?

- Users submit “jobs”
  - Each “job” contains 1+ “task” that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each “job” runs on one “cell”
  - A “cell” is a set of machines that run as one unit

- Two main types of jobs:
  - “Prod” job : long-running server jobs, higher priority
  - “Non-prod” job : quick batch jobs, lower priority

How does Borg work?

- **Allocs:**
  - Reserved set of resources in one machine
  - Can run multiple instances of a task, different tasks from many jobs, or future tasks
- **Priority and quota:**
  - Each job has a priority
  - Preemption disallowed between “prod” jobs.
  - Quota refers to vector of resource quantities for period of time
- **Support for naming and monitoring**

Borg Architecture - Scheduling

- Borgmaster adds new jobs to a pending queue after recording it in the Paxos store
- A scheduler (primarily operates on tasks) scans and assigns tasks to machines
  - Feasibility checking
  - Scoring
- E-PVM vs “best-fit”
  - E-PVM leaves headroom for load-spikes but has increased fragmentation
  - Best-fit fills machines as tightly as possible, but hurts “bursty loads”
- Current model is a hybrid of both
  - Borg will kill lower priority tasks until it finds room for an assigned task

Techniques Borg uses for managing utilization

- Cell-sharing: sharing prod and non-prod tasks
  - Resource reclaiming
  - Not sharing prod and non-prod work would increase machine needs by 20-30%
- Large cells: to allow large computations and decrease fragmentation
  - splitting up jobs and distributing them requires significantly more machines
- Fine-grained resource requests
  - fixed size containers/VMs not ideal
  - instead there are “buckets” of CPU/memory requirements
- Resource reclamation: jobs specify limits
  - Borg can kill tasks that use more RAM or disk space than requested
  - Throttle CPU usage
  - Prioritize prod tasks over non-prod

Borg Architecture - Scalability

- Ultimate scalability limit is unknown
  - Single Borgmaster can manage thousands of borglets
  - Rates above 10,000 tasks per minute
  - Busy Borgmaster uses 10-14 CPU cores and 50GiB RAM

Borg - Achieving Availability

- To mitigate inevitable failures, Borg will:
  - Automatically reschedule evicted tasks
  - Reduce correlated failures by distributing across failure domains
  - Limits downtime due to maintenance
  - Use “declarative desired-state representations and idem-potent mutating operations” to ease resubmission of forgotten requests
  - Avoid task to machine pairings that cause crashes
  - Use a logsaver to recover critical data written to a local disk
- Achieve 99.99% availability in practice

Isolation

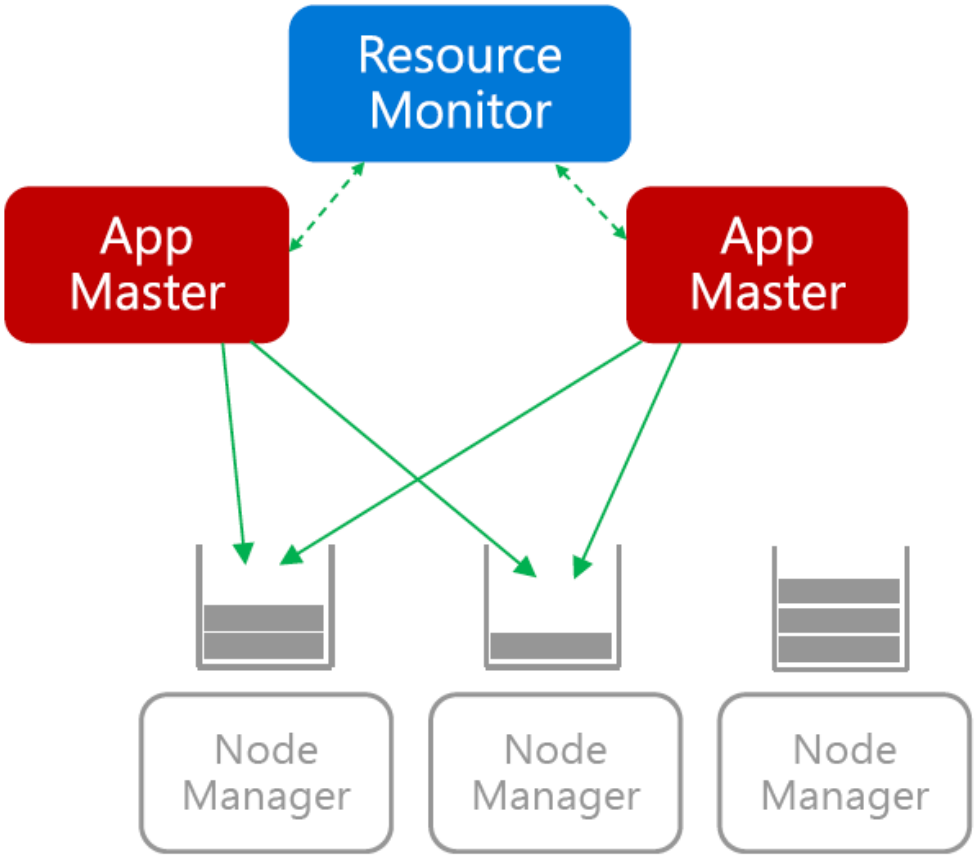
- Security:
  - Linux *chroot* command used for process isolation
  - Standard sandboxing techniques used for running external software
- Performance:
  - Borg makes explicit distinction between LS (latency-intensive) tasks and batch tasks. Helps for priority-based preemption
  - Borg uses notion of compressible resources (CPU cycles, disk I/O bandwidth) and non-compressible resources (RAM, disk space)

Why is it important to have isolation, and how does Borg implement it?

- To protect an app from Noisy, Nosy and Messy neighbors
- Sharing machines between applications increases utilization, but isolation is needed to prevent tasks from interfering
  - Security: rogue tasks can affect other tasks, and information should not be visible between tasks
  - Performance:
    - Utilization can be decreased by users inflating resource requests to prevent interference
    - Again, rogue tasks can affect your task
- Security: Linux chroot jail is the primary security isolation mechanism
- Performance: Linux cgroup-based container
  - Also appclass is used to help with overload and overcommitment
  - High priority LS (latency-sensitive) tasks

Approach 2: Distributed Resource Management

[Apollo, Sparrow]

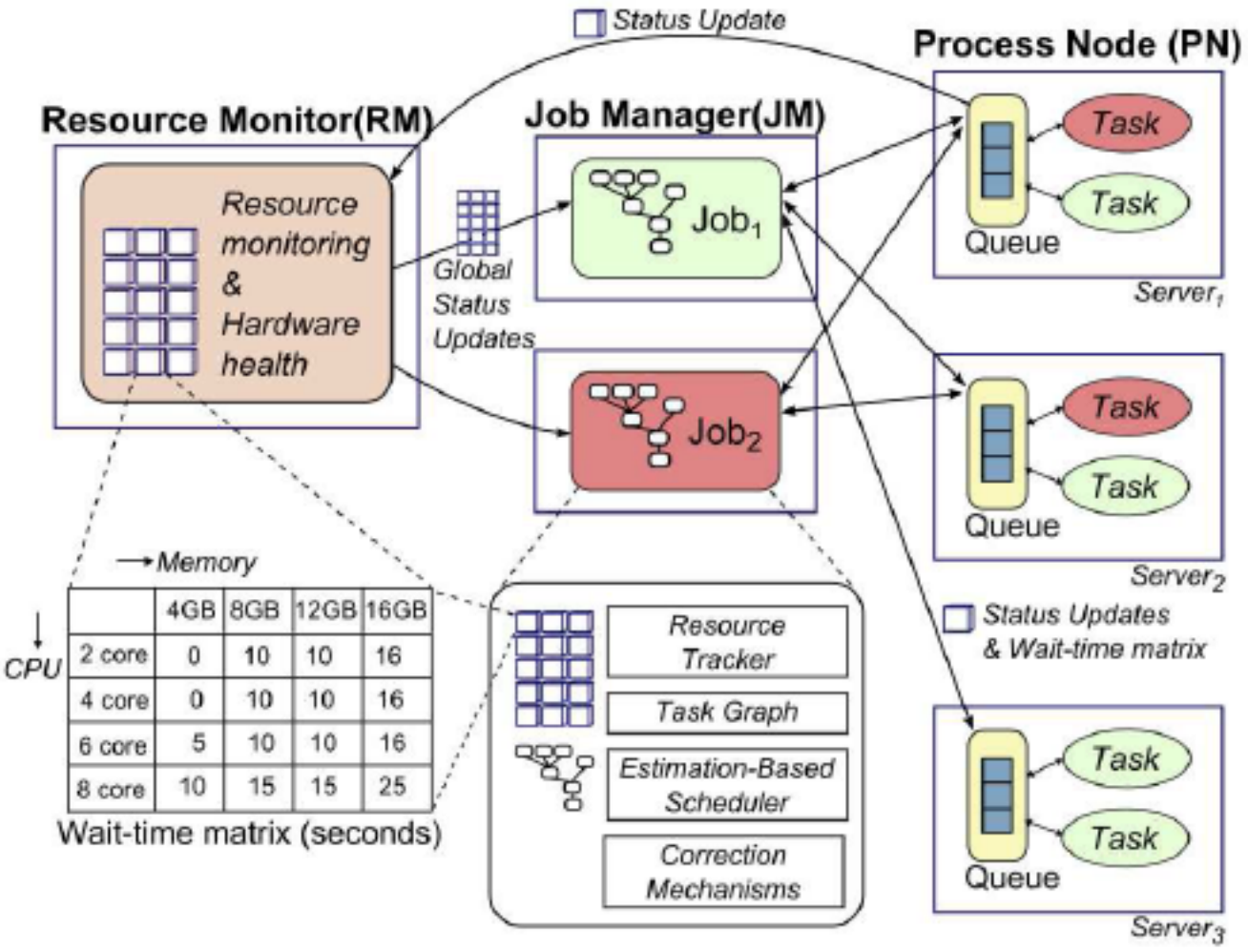


- AMs queue tasks directly to NMs
- Loose coordination through the Resource Monitor

K. Ousterhout et al, “Sparrow: Distributed, Low Latency Scheduling”, ACM SOSP 2013

E. Boutin et al, “Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing”, Usenix OSDI 2014

High-level Distributed Resource Management Architecture of Microsoft's Apollo



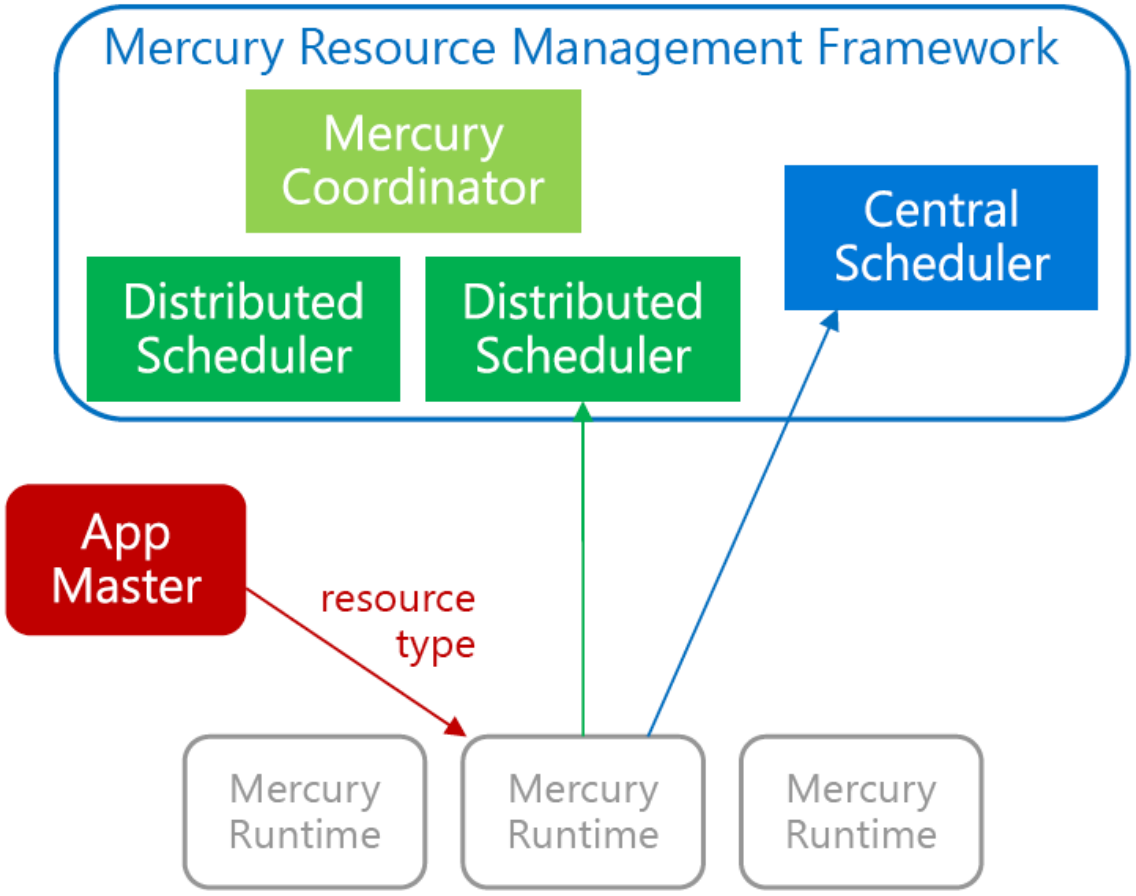
	4GB	8GB	12GB	16GB
2 core	0	10	10	16
4 core	0	10	10	16
6 core	5	10	10	16
8 core	10	15	15	25

E. Boutin et al, "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing", OSDI 2014

Centralized vs. Distributed Resource Management

	Centralized	Distributed
Workload heterogeneity	✓	
Task placement	✓	
Enforcing scheduling invariants	✓	
Allocation latency		✓
Slot utilization		✓
Scalability		✓

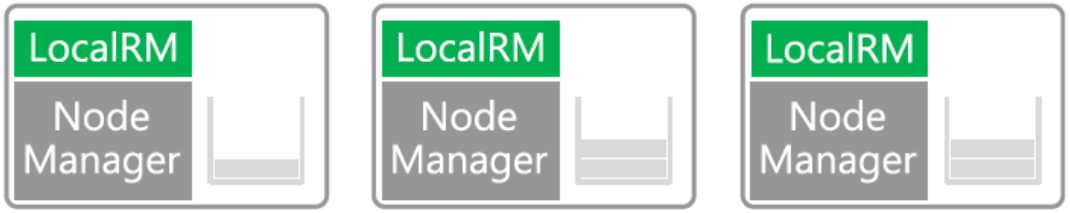
Approach 3: Hybrid (Distributed and Centralized) Resource Management in Microsoft's Mercury



- Two types of schedulers
- Central scheduler
  Scheduling policies/guarantees
  Slow(er) decisions
- Distributed schedulers
  Fast/low-latency decisions
- AM specifies resource *type*

K. Karanasos et al, "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters", Usenix ATC 2015

Mercury Architecture over YARN



Overview of YARN extensions

- LocalRM (distributed scheduling)
- Queuing of (QUEUEABLE) containers at the NMs
- Framework policies
- Application policies for determining container type per task

Operations and Implementation of Mercury

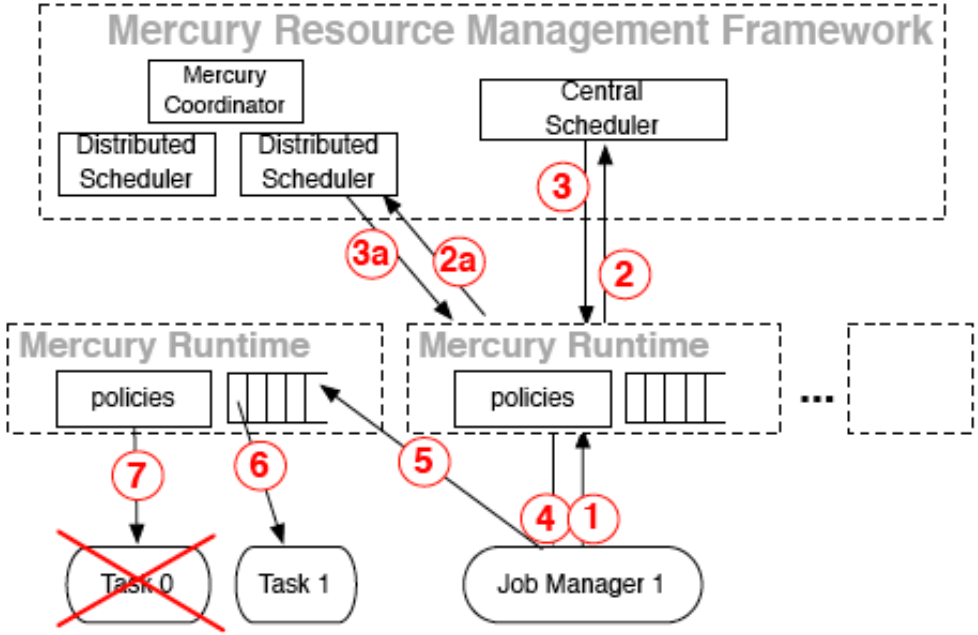


Figure 3: Mercury resource management lifecycle.

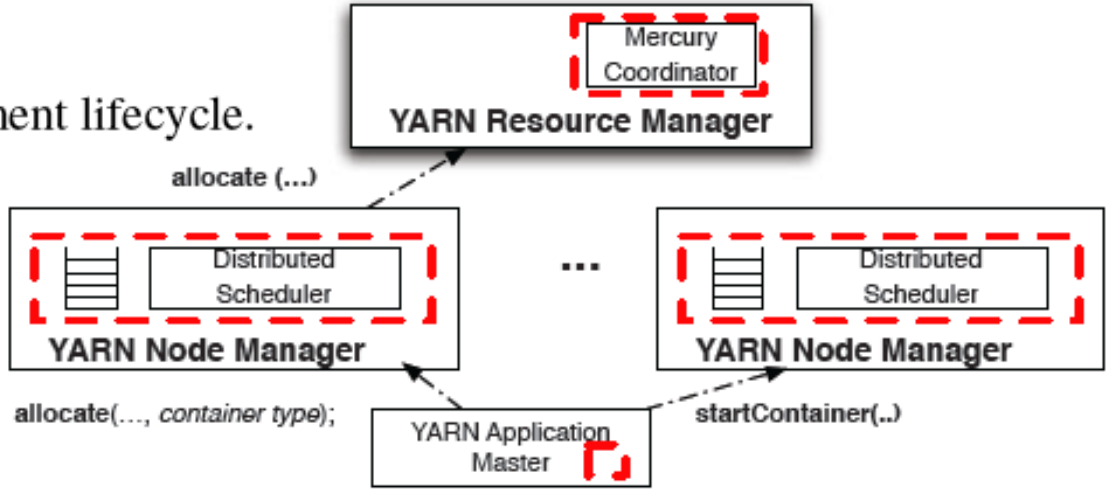


Figure 4: *Mercury implementation*: dashed boxes show Mercury modules and APIs as YARN extensions.

Comparisons of Recent Resource Management Platforms for Clusters

Resource Management Platform for Clusters	Scheduling/Resource Sharing paradigm	Scalability	Multiple Programming Frameworks/ Multi-tenant Support
Hadoop 1.0	Centralized	Limited but OK	No
YARN in Hadoop 2.0	Centralized	Good	Yes
Mesos	Centralized (Two-level) via Resource Offers to Individual Frameworks	Better	Yes
Apollo	Distributed and Loosely Coordinated (via Expected Resource Wait-Time matrix)	Very Good	Yes
Borg, Omega	Centralized per-cell BorgMaster which allows multiple // schedulers to performs optimistic-concurrent allocation followed by checking	Very Good	Yes
Mercury	Hybrid (Centralized and Distributed scheduling for Big and Small jobs respectively)	Very Good	Yes

Cloud-Native Applications (Micro-service oriented)

Cloud-Native Applications: Motivation

- Elasticity and Ubiquity of Cloud Infrastructure (Data-Center-scale Computing) have enabled new generation of applications, use-cases and business opportunities:
  - Netflix, Airbnb, Spotify, Pinterest, Snapchat, Whatspp,…
- Example: Netflix:
  - Value Proposition (Competitive Advantage):
    - Low Cost Video Streaming with Superb User-experience at scale
  - Application Properties and Unique Requirements
    - > 100 millions of users in 190 countries (mostly in US)
    - Vast variance in Load, within minutes (evenings, campaigns, …)
    - 10,000s of servers
    - 1000s of daily application changes across 100s of functions
      - Video streaming, Catalog, Recommendations, subscription
      - ~1 update every minute

Cloud-Native Applications: Requirements-driven Design Principles

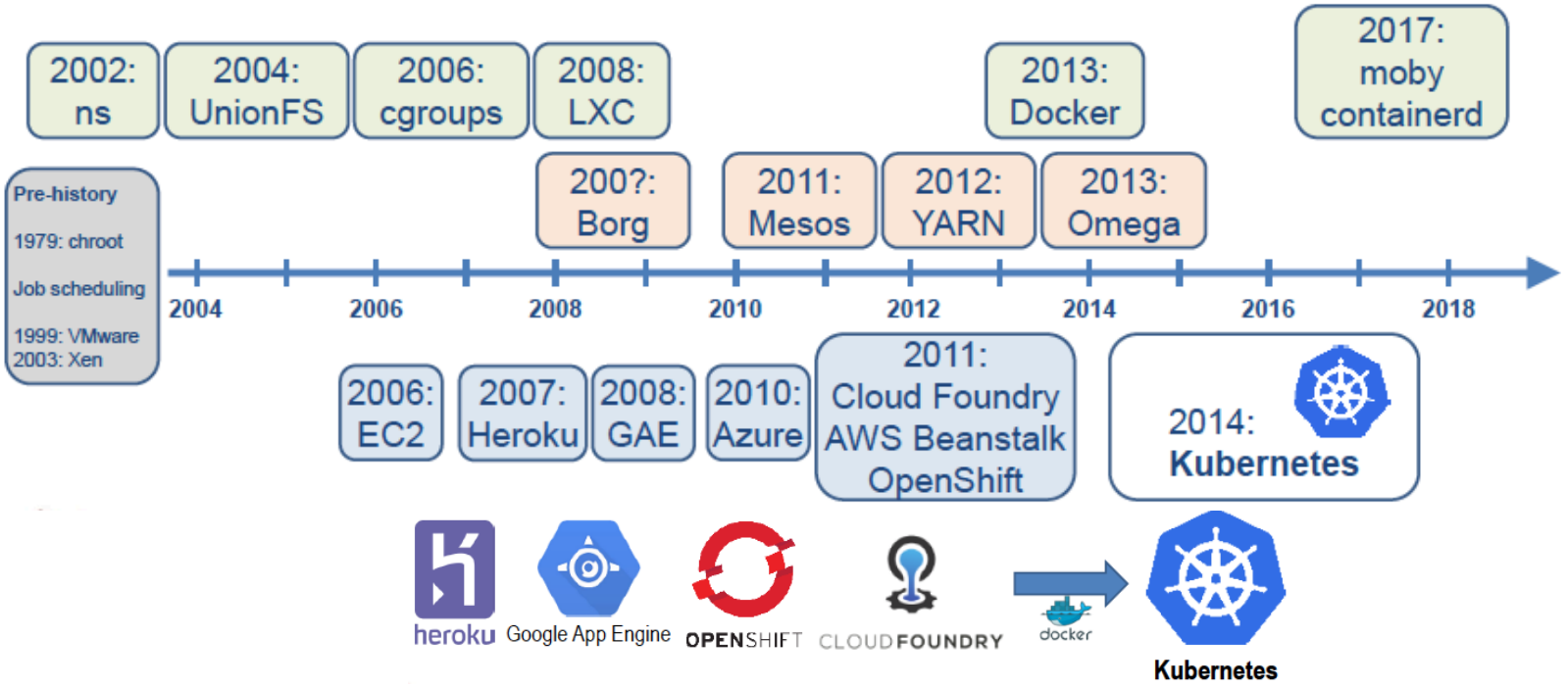
- Low Cost + Variance in Load + Superb User Experience
  - Can't afford Over or Under Provisioning => Auto-Scaling Elasticity
- Large-scale Infrastructure + Inevitable Hardware Failures + Superb User Experience
  - Must accommodate HW failures w/o downtime => Design for Failure
- Frequent Application Updates + Large-scale + Superb User Experience
  - Can't afford redeploying everything everytime => Modularity
  - Can't afford testing everything everytime => Stable Internal APIs
- Frequent Application Updates + Low Cost + Superb User Experience
  - Can't afford manual QA/ admin effort for each update => Automation in Deployment

Cloud-Native Applications: Design-driven Common Services

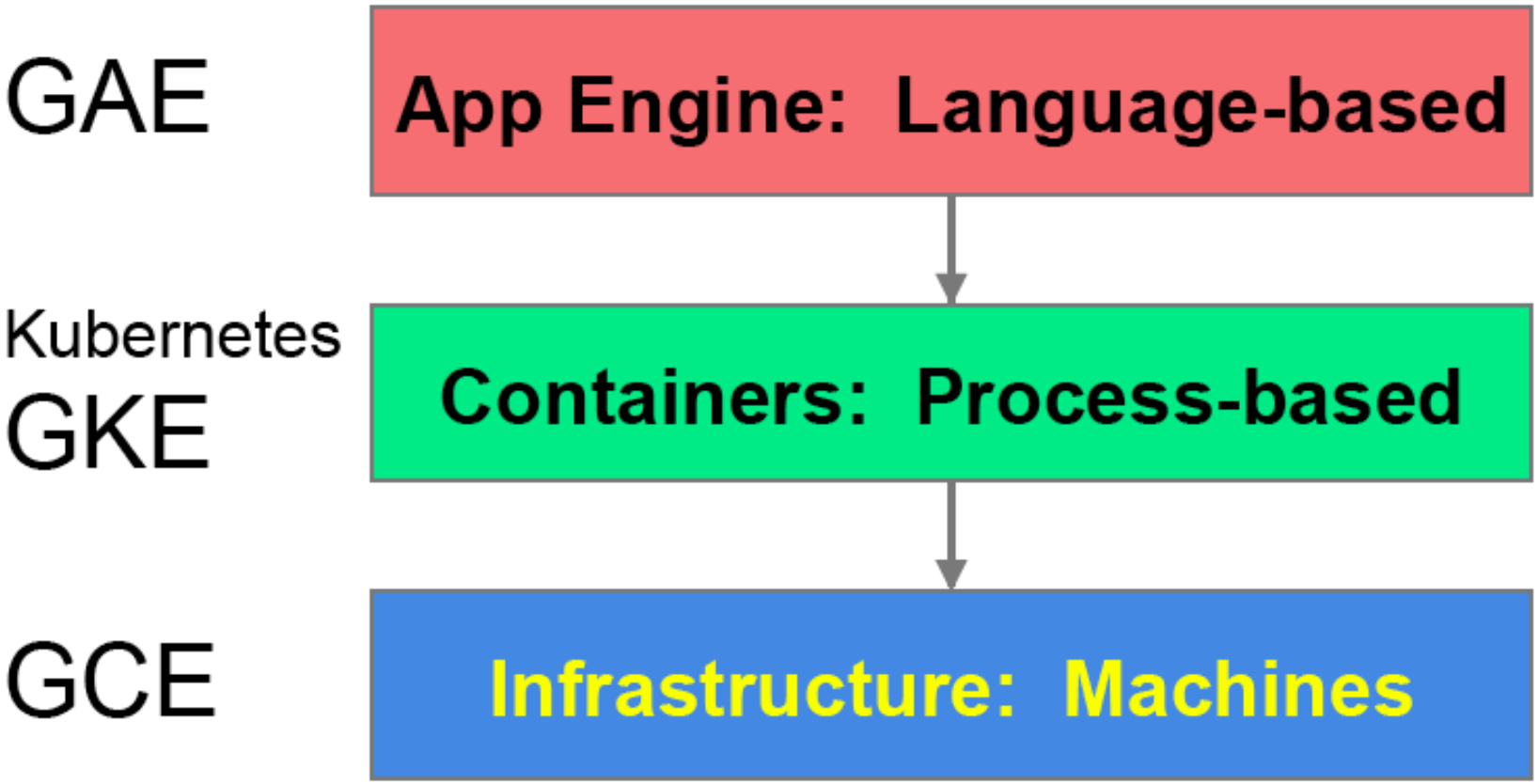
Design Principle	Common Service
Auto-Scaling	Horizontal Auto-Scaling
	Elastic Load Balancing
Design for Failure	Replication
	Health Monitoring
Modularity	Decoupling into homogenous (micro)services*
	Unified packaging (across dev/test/prod) with Docker
API-driven composition	Discovery
	Routing
Automation	Fully programmable life cycle of components*
	Observability (monitoring, tracing, etc)

Evolution of Platforms for Cloud-Native Applications:

- New Platforms emerged, offering common services required by Cloud-Native Applications:
  - Auto-Scaling, Replication, Load-Balancing, Health Monitoring, Service Discovery, Application-Level Routing, Programmability
  - Started in form of “Platform as a Service” (PaaS)
  - Eventually generalized to the Container-based Orchestration approach



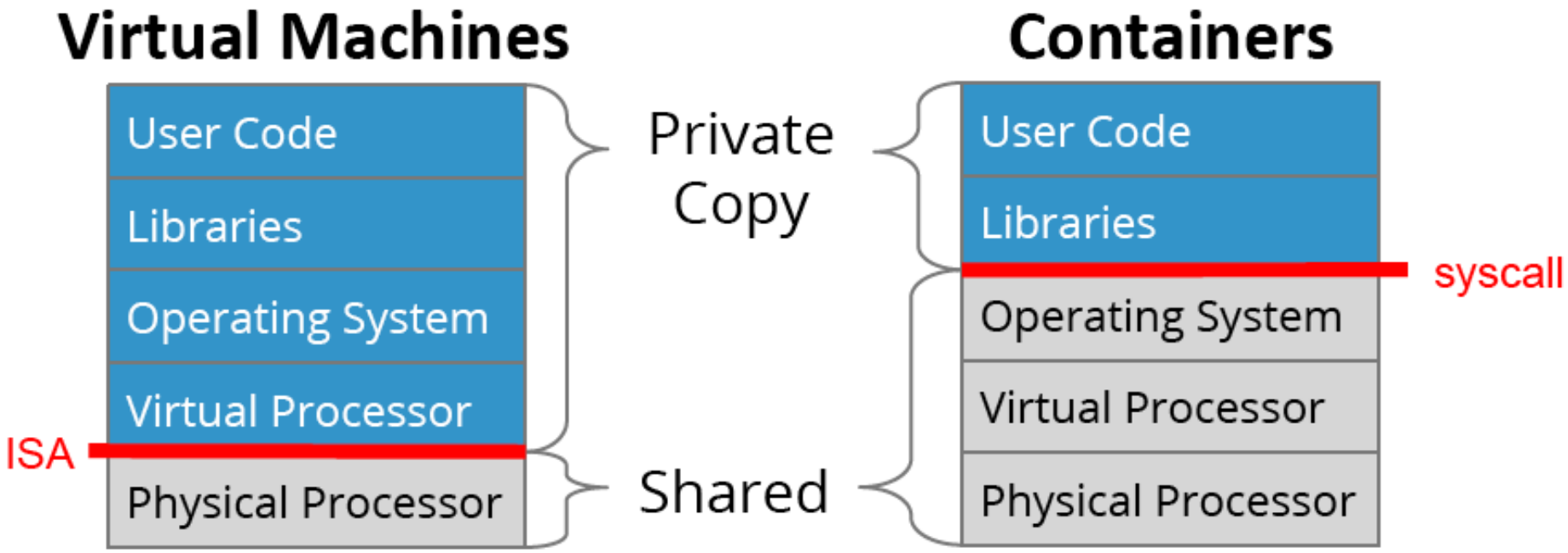
Different forms of Cloud-based Computing Services/Offerings from Google



Deploying Cloud-Native (Micro-service oriented) Applications

Container Technologies to our rescue !

VMs vs. Containers



Containers: less overhead, enable more "magic"

Virtual Machine vs. Container

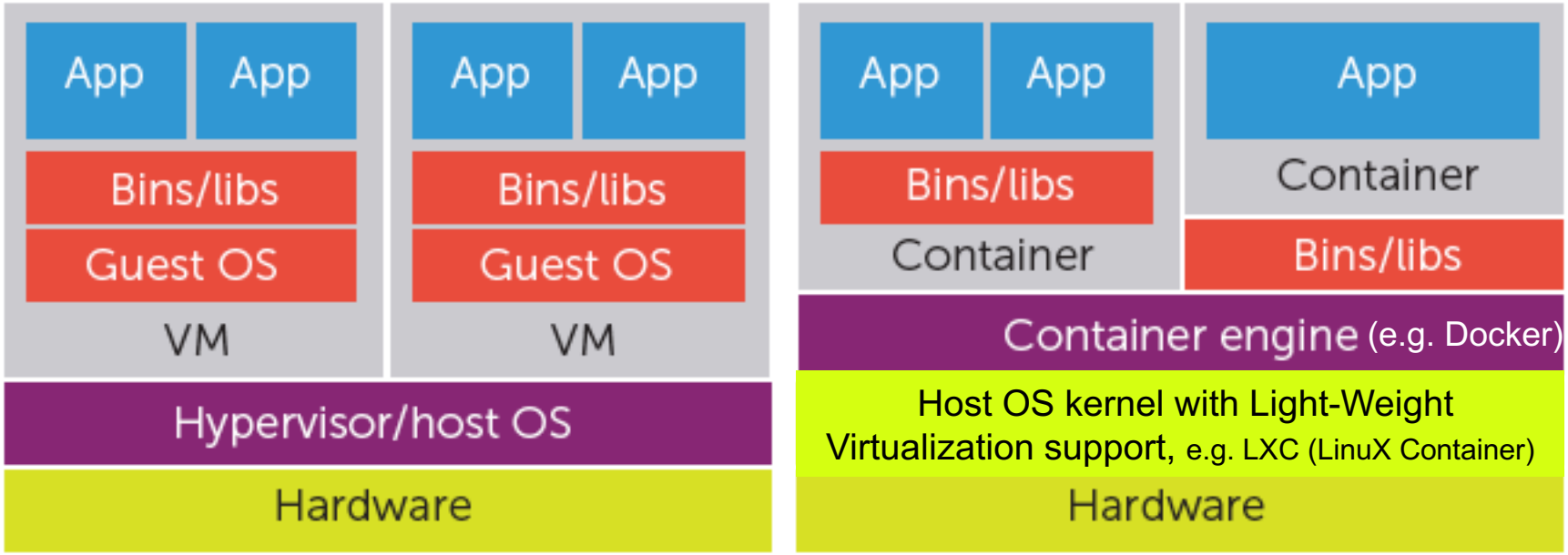
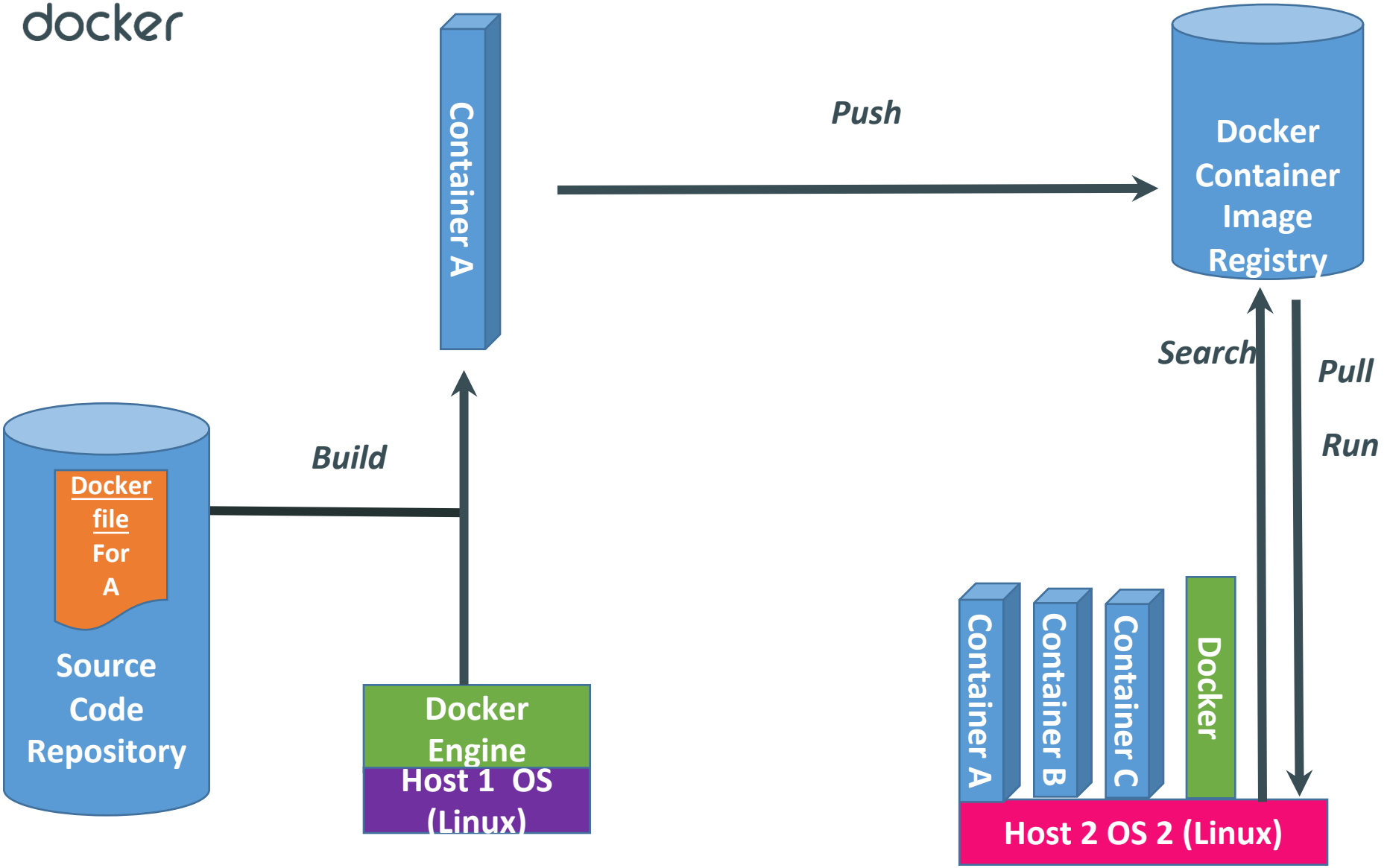


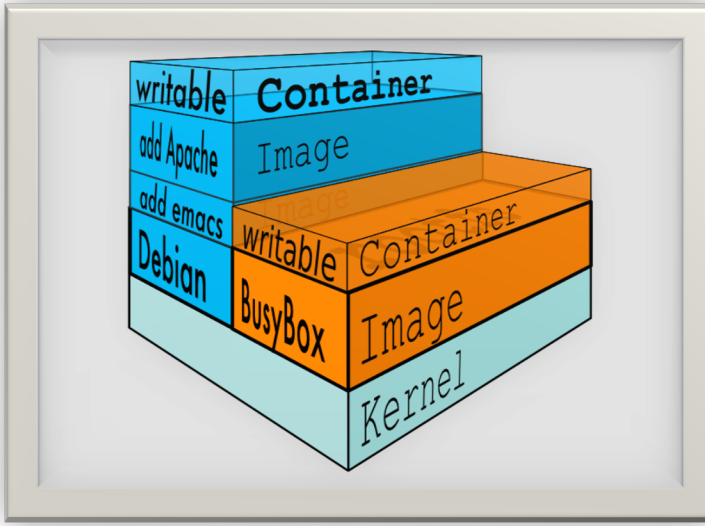
FIGURE 1. Virtualization architecture. The two possible scenarios, a traditional hypervisor architecture on the left and a container-based architecture on the right, differ in their management of guest operating system components.

Source: Claus Pahl, "Containerization and the PaaS Cloud," IEEE Cloud Computing Magazine, May/June 2015

Basic Operations of a Docker system



Docker Containers

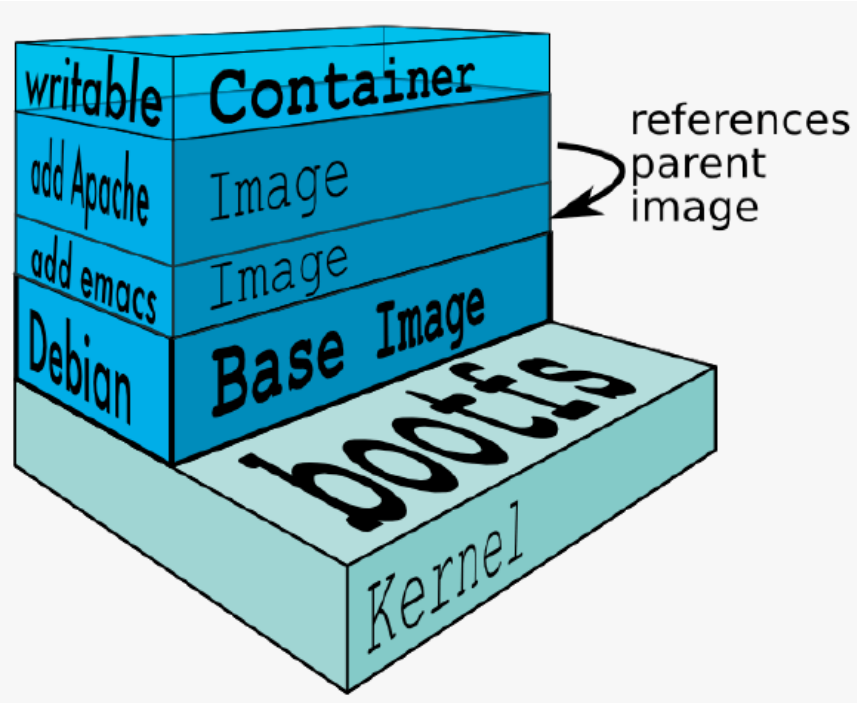


- Units of software delivery (ship it!)
  - run everywhere
    - regardless of kernel version
    - regardless of host distribution
    - (but container and host architecture must match*)
  - run anything
    - if it can run on the host, it can run in the container
    - i.e., if it can run on a Linux kernel, it can run

*Unless you emulate CPU with QEMU and binfmt

Docker Image structure

- NOT A Virtual Hark Disk (VHD) file
- NOT A FILESYSTEM
- uses a *Union File System*
- a read-only
- do not have state
- Basically a tar file
- Has a hierarchy
  - Arbitrary depth
- Fits into the Docker Registry



Google's Kubernetes: - Merging 2 Different Types of Containers

Docker

- It's about ***packaging***
- Control:
  - packages
  - versions
  - (some config)
- Layered file system
- ⇒ Prod matches testing

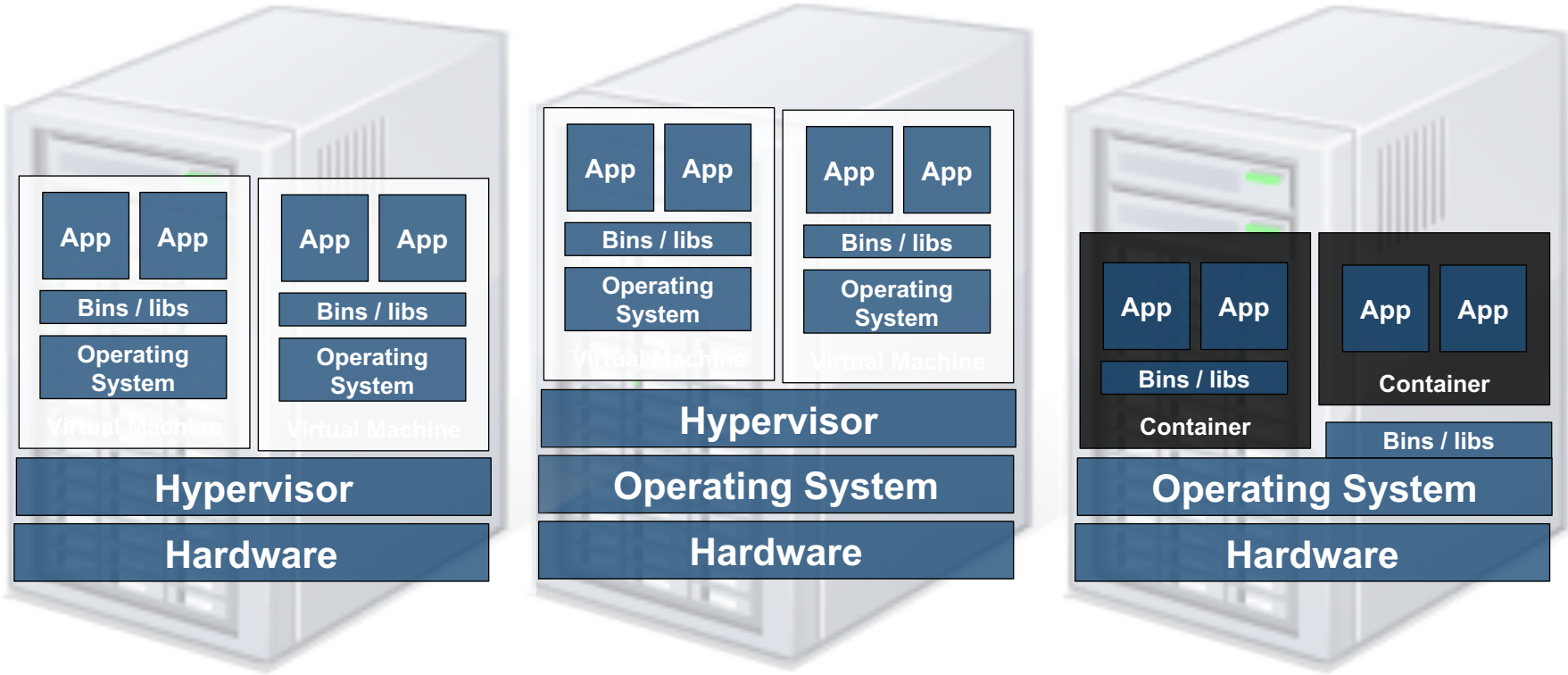
Linux Containers

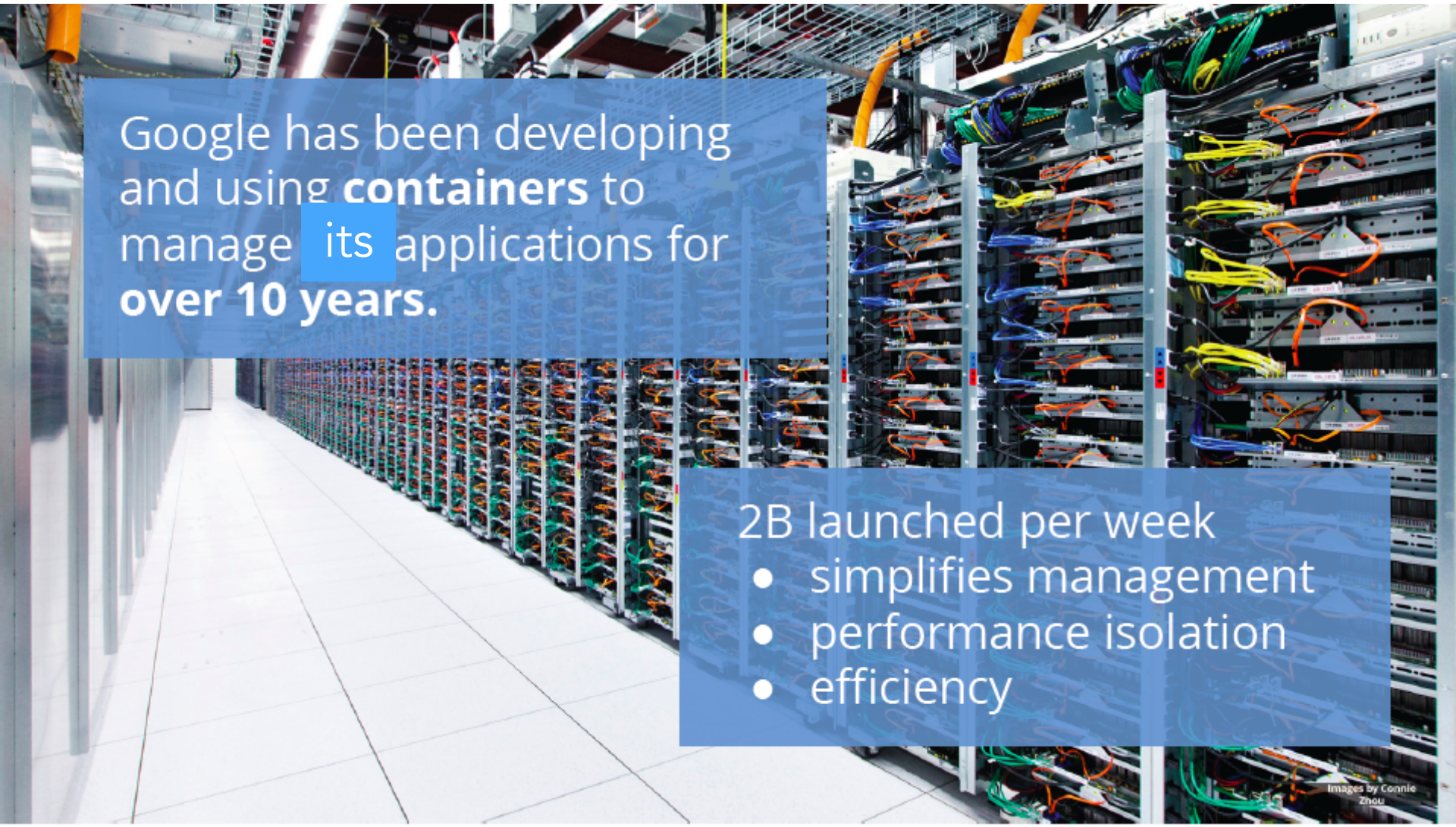
- It's about ***isolation*** … *performance isolation*
- not *security* isolation … use VMs for that
- Manage CPUs, memory, bandwidth, …
- Nested groups

Hypervisors vs. Linux Containers

Conceptual Mapping

VM → Container
Hypervisor → LXC Engine




Google has been developing and using **containers** to manage its applications for **over 10 years.**
2B launched per week
- simplifies management
- performance isolation
- efficiency
Images by Connie Zhou

Why Containers ?

- Ease of development:
  - User makes jobs based on containers ; the cluster/cloud schedule those jobs for them
    - User don't need to worry about machines or the OS
- High Resource Utilization: (more efficient)
  - The scheduler can pack many containers per machine
  - Can mix Live-services and Batch workload
    - Use Batch-job to fill in the holes
- Ease of Operation: (fewer staff per job)
  - e.g. All jobs use latest security patches
  - More shared code among projects (shared services and code)

Kubernetes (K8s)

κυβερνήτης: Greek for “pilot” or “helmsman of a ship”

The open-source cluster manager from Google

- Container Orchestrator
- Run Docker Container
- Support multiple cloud and bare-metal environments
- Inspired and informed by Google’s experiences & internal systems, e.g. the Borg scheduler
- Open source, written in Go:
  - Google has donated it to Cloud Native Computing Forum (CNCF)

Key: Kubernetes manages Applications NOT Machines

Kubernetes: Higher Level of Abstraction

Think About

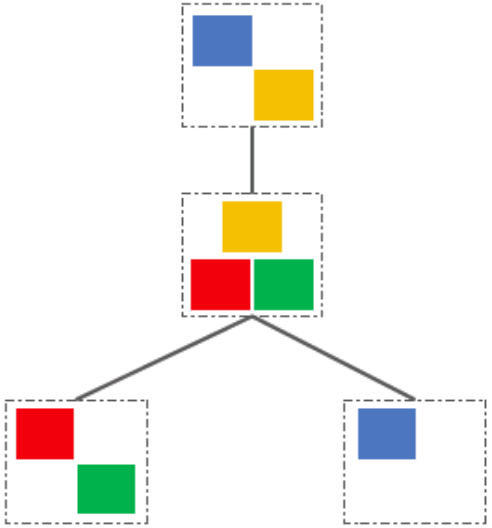
- Composition of services
- Load-balancing
- Names of services
- State management
- Monitoring and Logging
- Upgrading

Don't Worry About

- OS details
- Packages — no conflicts
- Machine sizes (much)
- Mixing languages
- Port conflicts

Kubernetes – Google's path towards "Cloud-native" Applications

- Kubernetes serves as a distributed platform for Hosting and Orchestrating Containers in a clustered environment
  - Support: Container Grouping (Pods), Replication, Scheduling, Load-Balancing, Auto-Healing, Scaling, Service Discovery, etc .
- Cloud-native Apps often structured as Interacting Microservices
  - Encapsulated states with APIs, like "Objects"
  - Mix of Programming Languages
  - Mix of Teams



Don't think of a container as the boundary of your application

"A container is more like a class in an object-oriented language."

--- Google's Brendan Burns

Services Model

- Each app lives in an environment of shared services
  - Storage, monitoring, logging
  - Master election, deployment, testing
- Services are *Abstract*
  - A “Service” is just a long-lived abstract name
  - Varied Implementations over time (versions)
    - Multiple versions running at a time
      - Required for “canary” testing -- deploy new version bit-by-bit gradually
    - Usually new versions are backward compatible
    - Sometimes not => must eventually update ALL clients
    - Running multiple versions gives clients/users some time to update
  - Services can be (and often are) updated independently
- Kubernetes routes to the right implementation

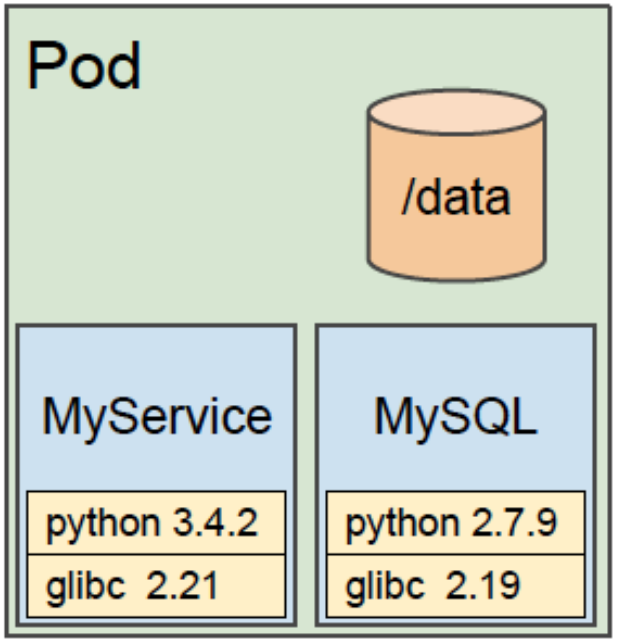
Managing Dependencies w/ K8s

Containers:

- Handle *package* dependencies
- Different versions, same machine
- No “DLL hell”

Pods:

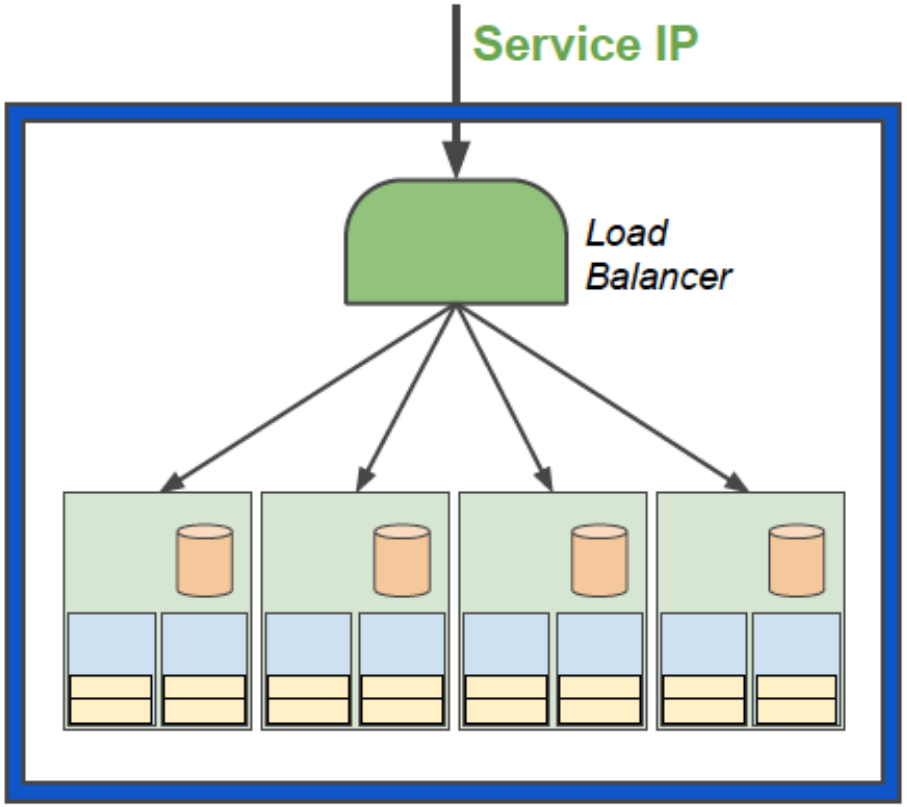
- *Co-locate* containers
- Shared volumes
- IP address, independent port space
- Unit of deployment, migration



Running/ Managing Services w/ K8s

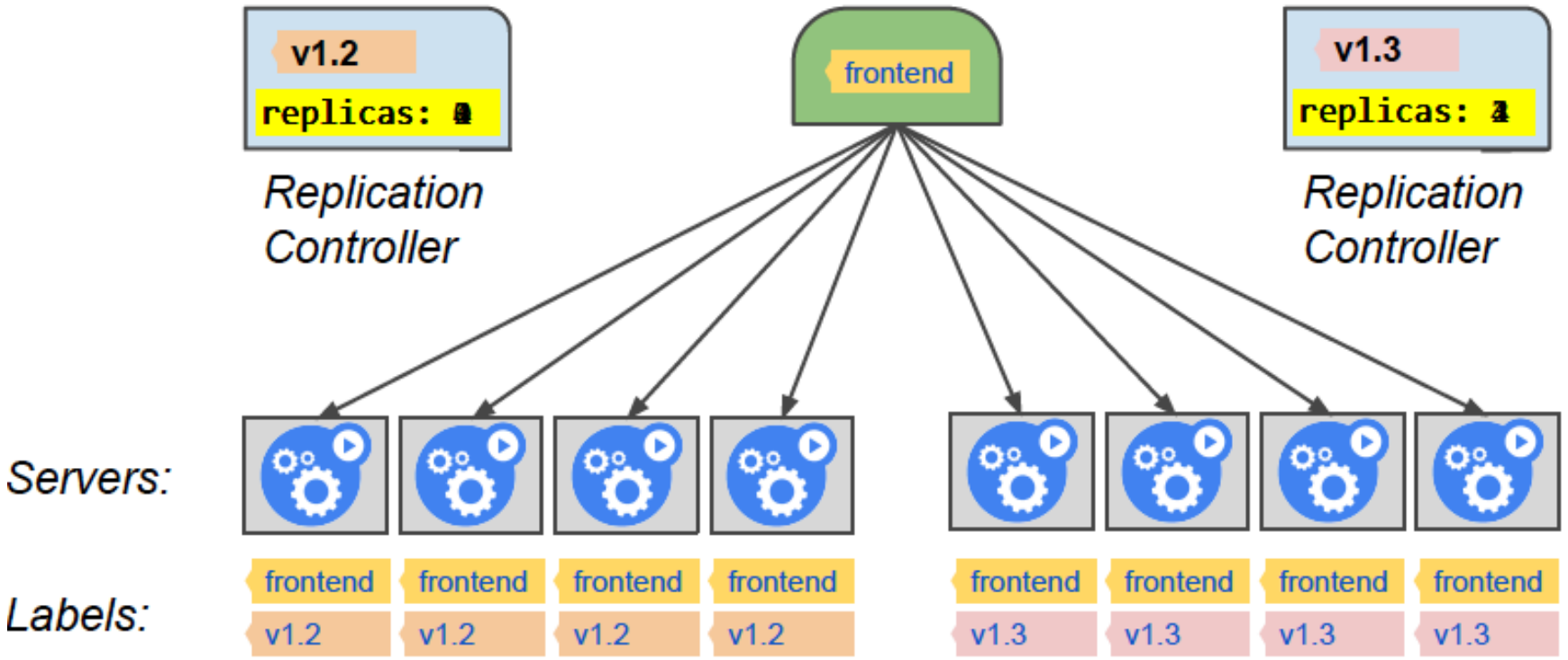
Services:

- Replicated pods
  - Source pod is a template
- Auto-restart member pods
- Abstract name (DNS)
- IP address for the service
  - in addition to the members
- Load balancing among replicas

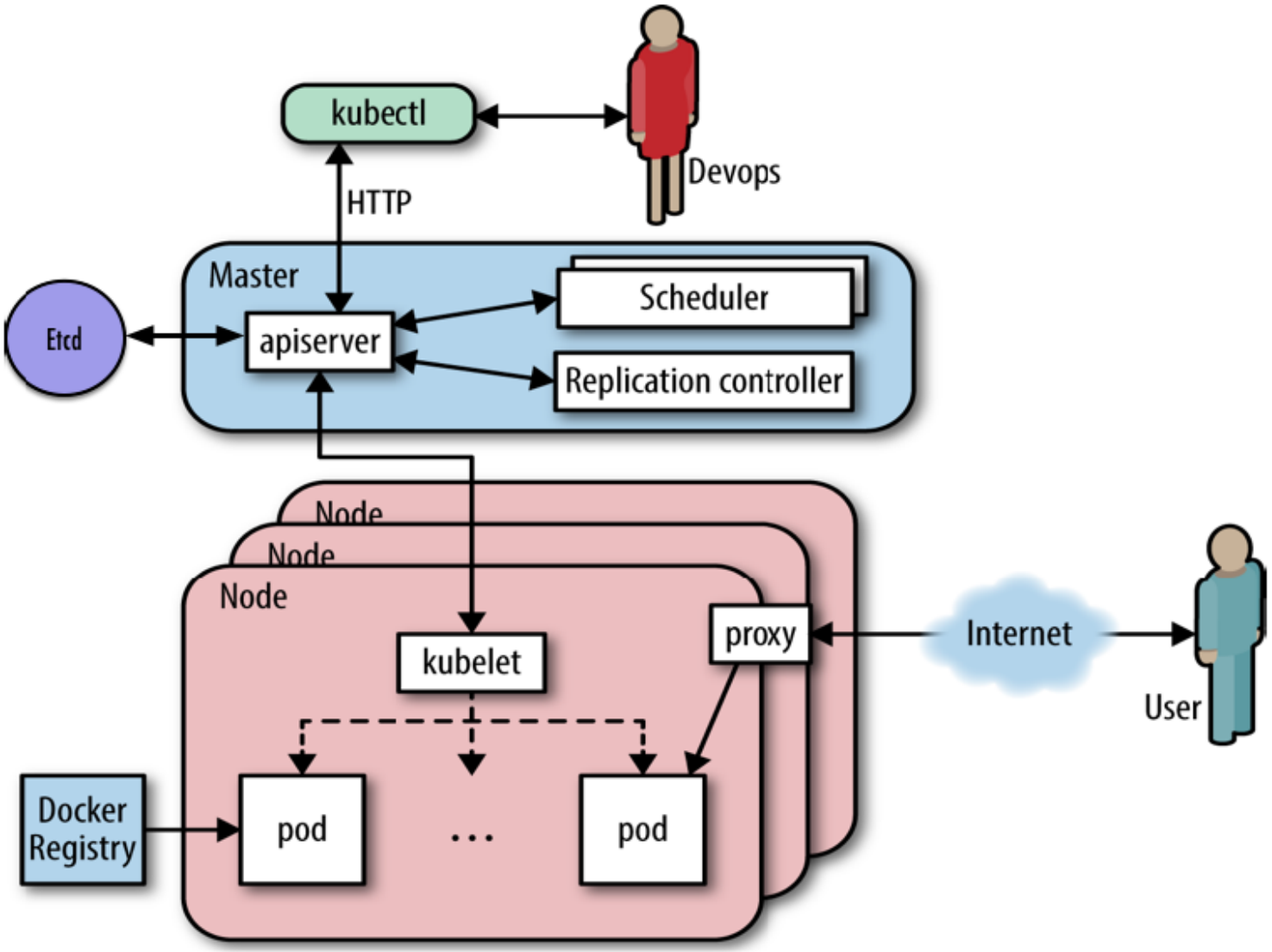


Managing Service Dependenices w/ K8s

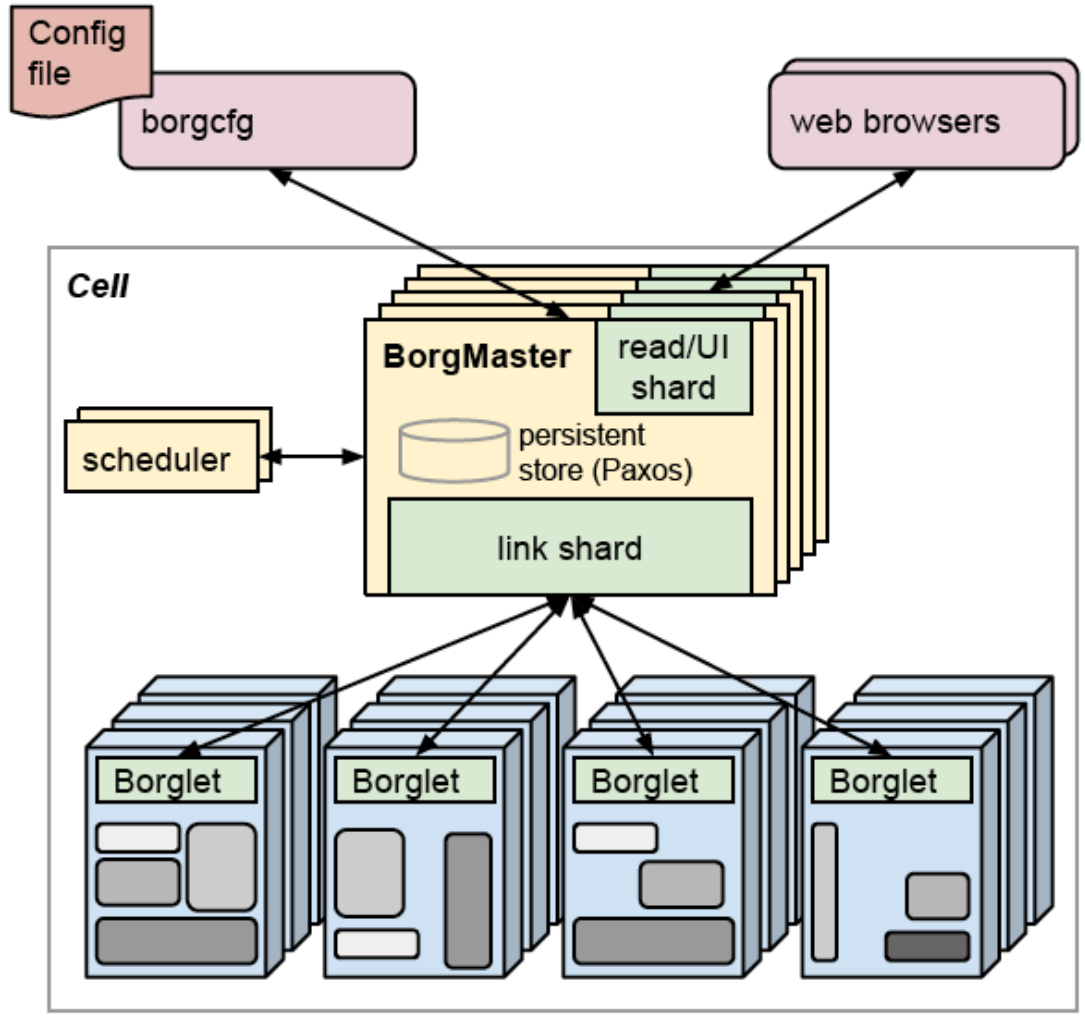
Example: Rolling Upgrade with Labels



System Architecture of Kubernetes



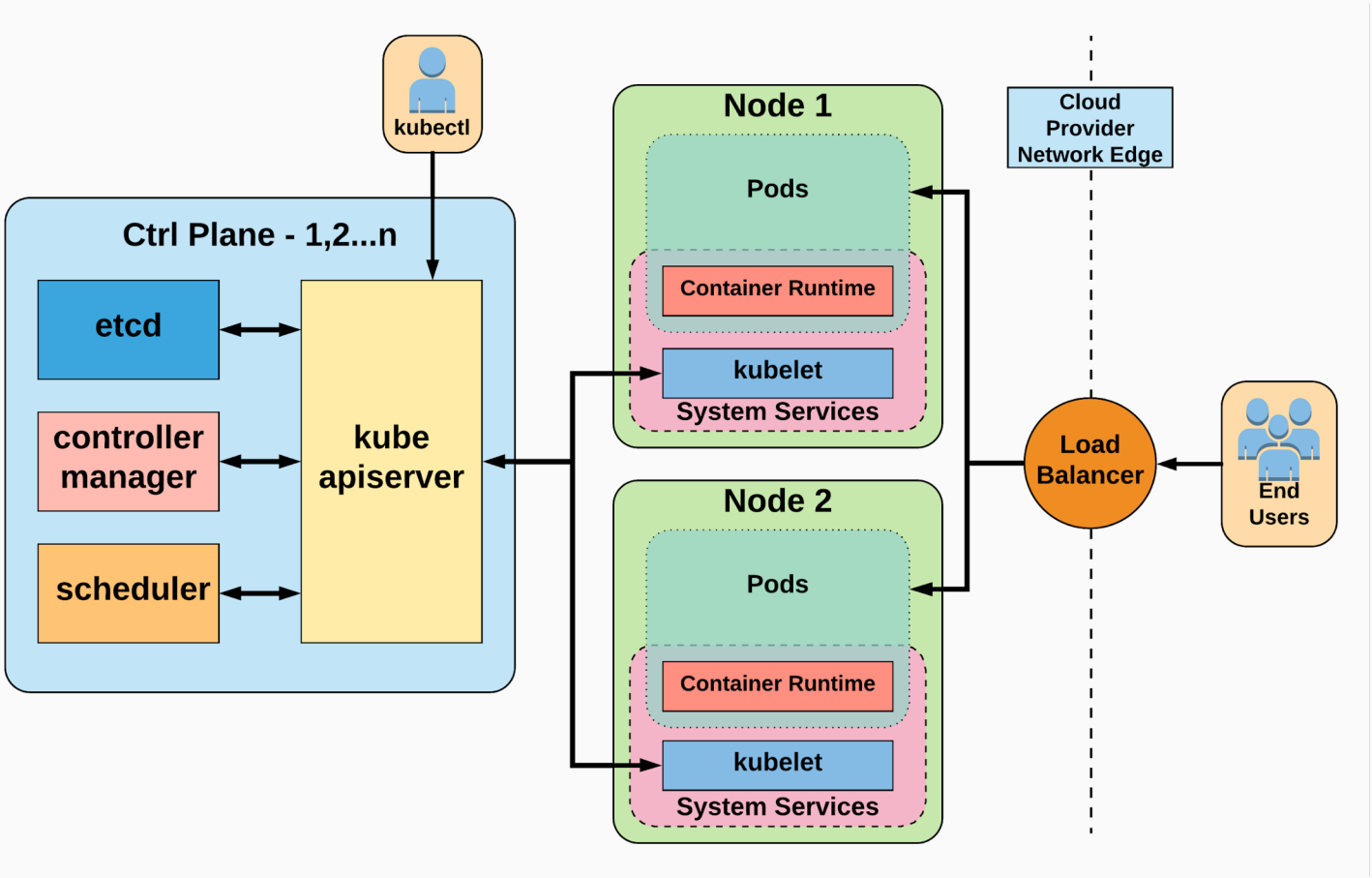
Recall - The Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Kubernetes Architecture

Kubernetes Architecture Overview



Source: Introduction to Kubernetes

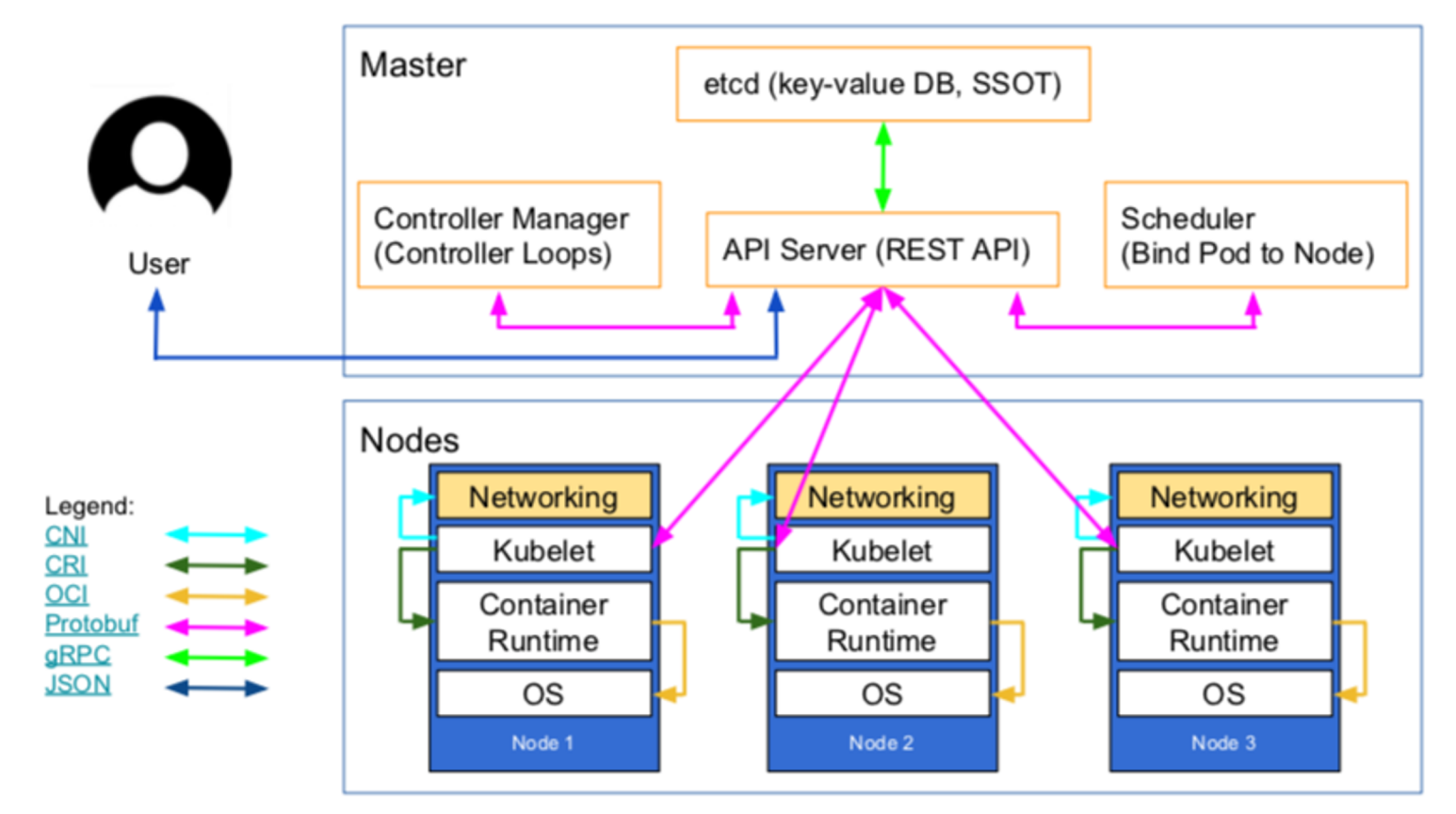
50K feet on how Kubernetes works

How Kubernetes works?

In Kubernetes, there is one (or more) master node and multiple worker nodes, each worker node can handle multiple pods. Pods are just a bunch of containers clustered together as a working unit. You can start designing your applications using pods. Once your pods are ready, you can specify pod definitions to the master node, and how many you want to deploy. From this point, Kubernetes is in control. It takes the pods and deploys them to the worker nods.

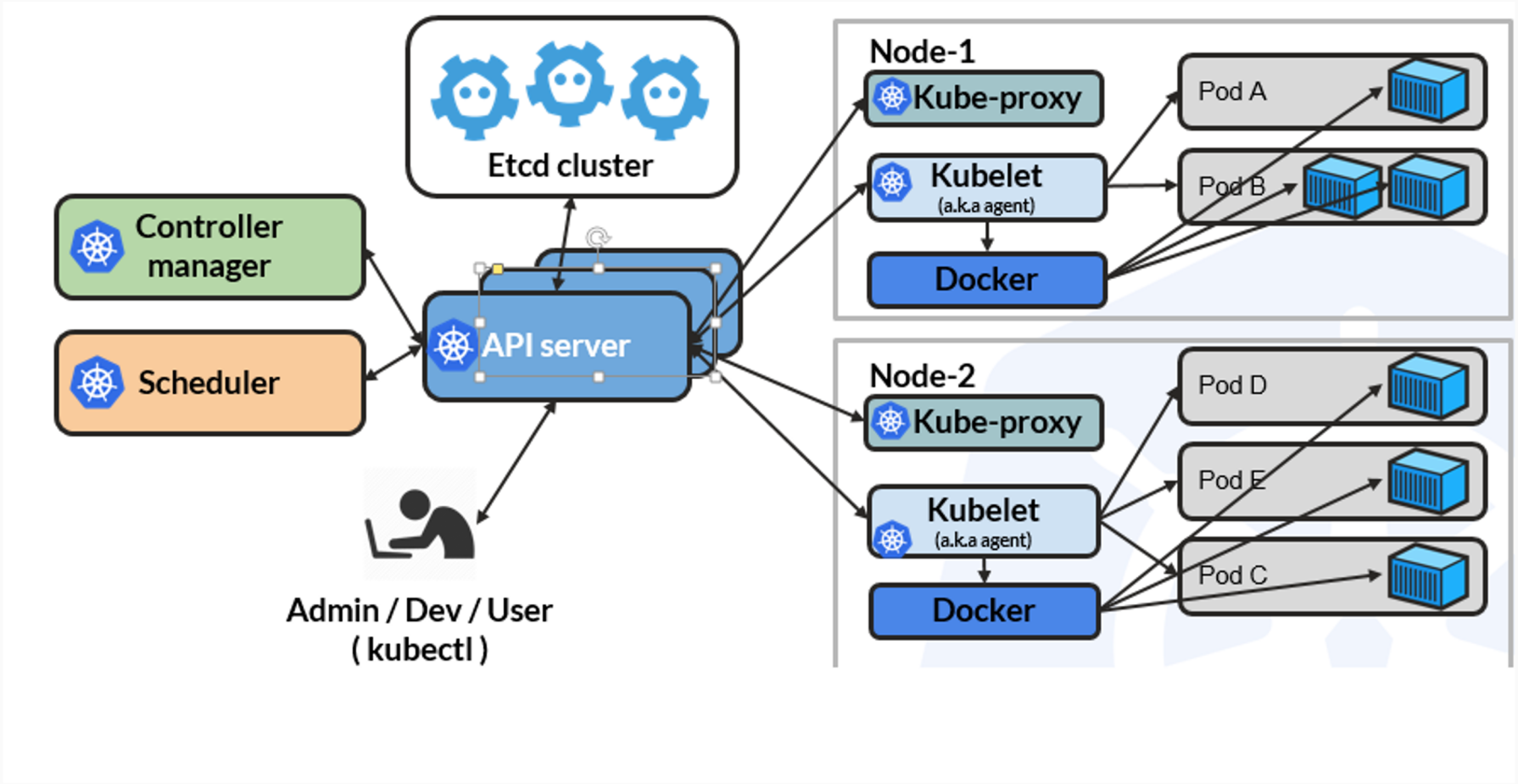
Source: https://itnext.io/successful-short-kubernetes-stories-for-devops-architects-677f8bfed803

Kubernetes' High-Level Architecture Overview



Source: https://www.weave.works/blog/what-does-production-ready-really-mean-for-a-kubernetes-cluster

Kubernetes' High-Level Architecture Overview



Core Concepts of Kubernetes

Cluster — Set of machines where pods are deployed, managed and scaled.

- Nodes are connected via a "flat" network.
- Typical cluster sizes range from 1-200 nodes.

Pod — A pod consists of one or more containers that are guaranteed to be co-located on the same machine.

- Share storage volumes and a networking stack.
- A pod is the basic unit of scheduling.

Controller — A controller is a reconciliation loop that drives actual cluster state toward the desired cluster state.

- **Replication Controller** — Handles replication and scaling by running a specified number of copies of a pod across the cluster.

Service — Set of pods that work together, such as one tier of a multi-tier application. Kubernetes provides:

- Service discovery
- Request routing by assigning a stable IP address and DNS name
- Load balancing

Label — The user can assign key-value pairs (called labels) to any API object in the system (e.g., pods, nodes).

- **Label selector** — A query against a label that returns matching objects.

Borg vs. Kubernetes Comparisons:

- Borg is a predecessor to Kubernetes
- Borg groups tasks by 'job' and simple numeric index; Kubernetes adds 'labels' for greater flexibility.
- Kubernetes allows for Docker and other containers, while Borg only allows LMCTFY
- Borg exposes the API of its various components to allow external program to access/control cluster-resource directly ; All accesses to a K8s-cluster must go through a single-point-of-contact: the API-server
- Single IP per machine in Borg complicates things
  - Because of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes assigns every "pod" and service its own IP address
  - Allows developers to choose ports and removes the infrastructure complexity of managing ports
- Borg is not open source or available for use outside of Google unlike Kubernetes
  - Both work on bare metal, but Kubernetes can work on various cloud hosting providers, "such as Google Compute Engine."

More Concepts of Kubernetes

- **Controllers**
  - **Deployments →**
  - **ReplicaSet →**
  - **ReplicationController →**
  - **DaemonSet →**
- **StatefulSets →**
- **ConfigMaps →**
- **Secrets →**
- **Persistent Volumes (attaching storage to containers) →**
- **Life Cycle of Applications in Kubernetes →**
  - **Updating Pods**
  - **Rolling updates**
  - **Rollback**

What is a Deployment in K8s ?

- A Deployment provides declarative updates for Pods and ReplicaSets, specifying the “desired” state of a deployment
  - The Deployment Controller is responsible to change the actual state to the desired state. This is the so-called “reconciliation” process.

Use Case of Deployments:

- Create a Deployment to rollout a ReplicaSet
- Declare the new state of the Pods
- Rollback to an earlier Deployment revision
- Scale up the Deployment to facilitate more Load
- Pause for Deployment to apply fixes to its PodTemplateSpec before resume it to start a new rollout

Kubernetes Resources Explained (1)

Kubernetes resources explained (1)		
	Resource (abbr.) [API version]	Description
	Namespace* (ns) [v1]	Enables organizing resources into non-overlapping groups (for example, per tenant)
Deploying Workloads	Pod (po) [v1]	The basic (atomic) deployable unit containing one or more processes in co-located containers
	ReplicaSet	Keeps one or more pod replicas running
	ReplicationController	The older, less-powerful equivalent of a ReplicaSet
	Job	Runs pods that perform a completable task
	CronJob	Runs a scheduled job once or periodically
	DaemonSet	Runs one pod replica per node (on all nodes or only on those matching a node selector)
	StatefulSet	Runs stateful pods with a stable identity
	Deployment	Declarative deployment and updates of pods

Source: Kubernetes in Action book by Marko Lukša

Kubernetes Resources Explained (2)

	Resource (abbr.) [API version]	Description
Services	Service (svc) [v1]	Exposes one or more pods at a single and stable IP address and port pair
	Endpoints (ep) [v1]	Defines which pods (or other servers) are exposed through a service
	Ingress (ing) [extensions/v1beta1]	Exposes one or more services to external clients through a single externally reachable IP address
Config	ConfigMap (cm) [v1]	A key-value map for storing non-sensitive config options for apps and exposing it to them
	Secret [v1]	Like a ConfigMap, but for sensitive data
Storage	PersistentVolume* (pv) [v1]	Points to persistent storage that can be mounted into a pod through a PersistentVolumeClaim
	PersistentVolumeClaim (pvc) [v1]	A request for and claim to a PersistentVolume
	StorageClass* (sc) [storage.k8s.io/v1]	Defines the type of storage in a PersistentVolumeClaim

Source: Kubernetes in Action book by Marko Lukša

Kubernetes Resources Explained (3)

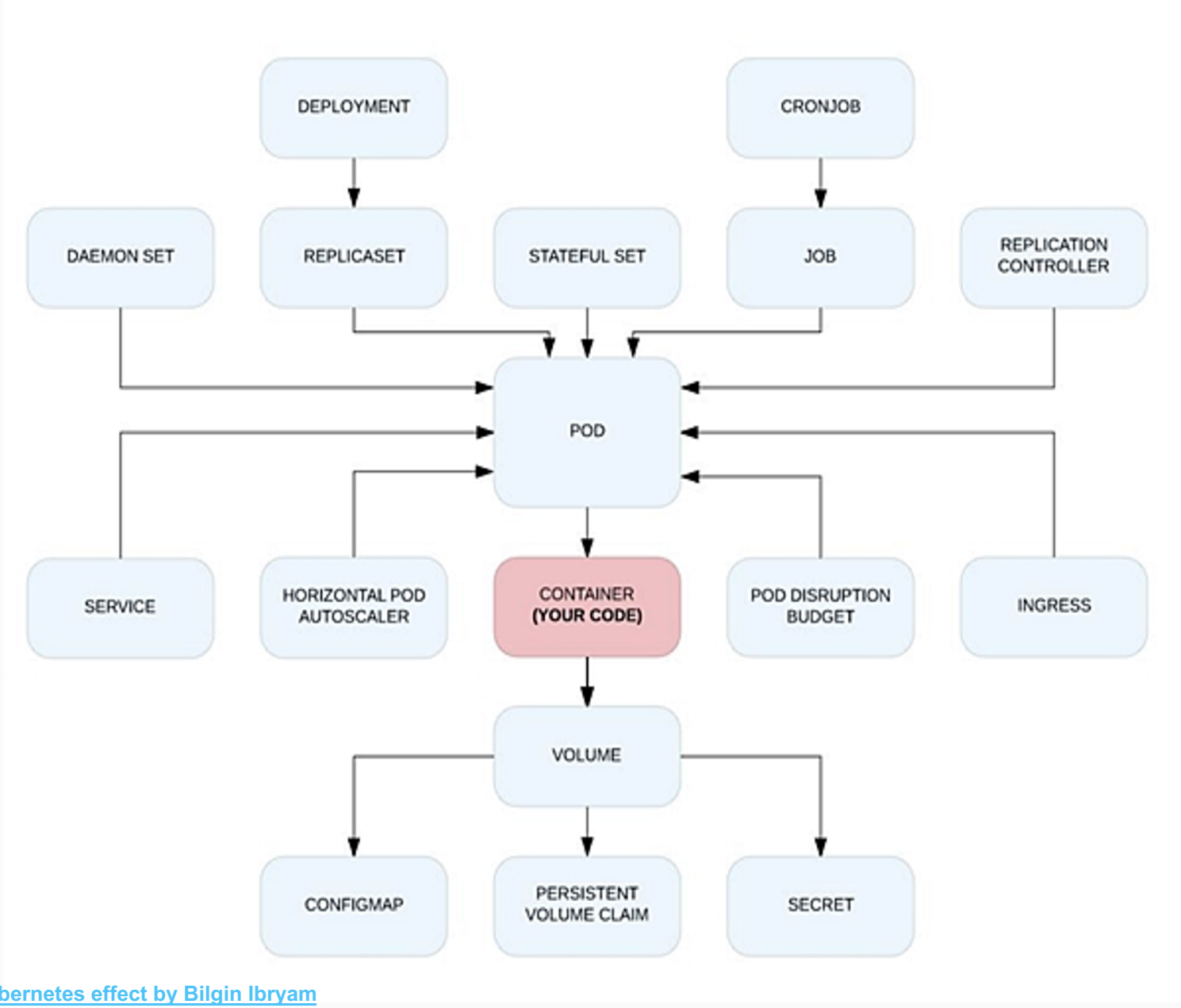
	Resource (abbr.) [API version]	Description
Scaling	HorizontalPodAutoscaler (hpa) [autoscaling/v2beta1**]	Automatically scales number of pod replicas based on CPU usage or another metric
	PodDisruptionBudget (pdb) [policy/v1beta1]	Defines the minimum number of pods that must remain running when evacuating nodes
Resources	LimitRange (limits) [v1]	Defines the min, max, default limits, and default requests for pods in a namespace
	ResourceQuota (quota) [v1]	Defines the amount of computational resources available to pods in the namespace
Cluster state	Node* (no) [v1]	Represents a Kubernetes worker node
	Cluster* [federation/v1beta1]	A Kubernetes cluster (used in cluster federation)
	ComponentStatus* (cs) [v1]	Status of a Control Plane component
	Event (ev) [v1]	A report of something that occurred in the cluster

Source: Kubernetes in Action book by Marko Lukša

Kubernetes Resources Explained (4)

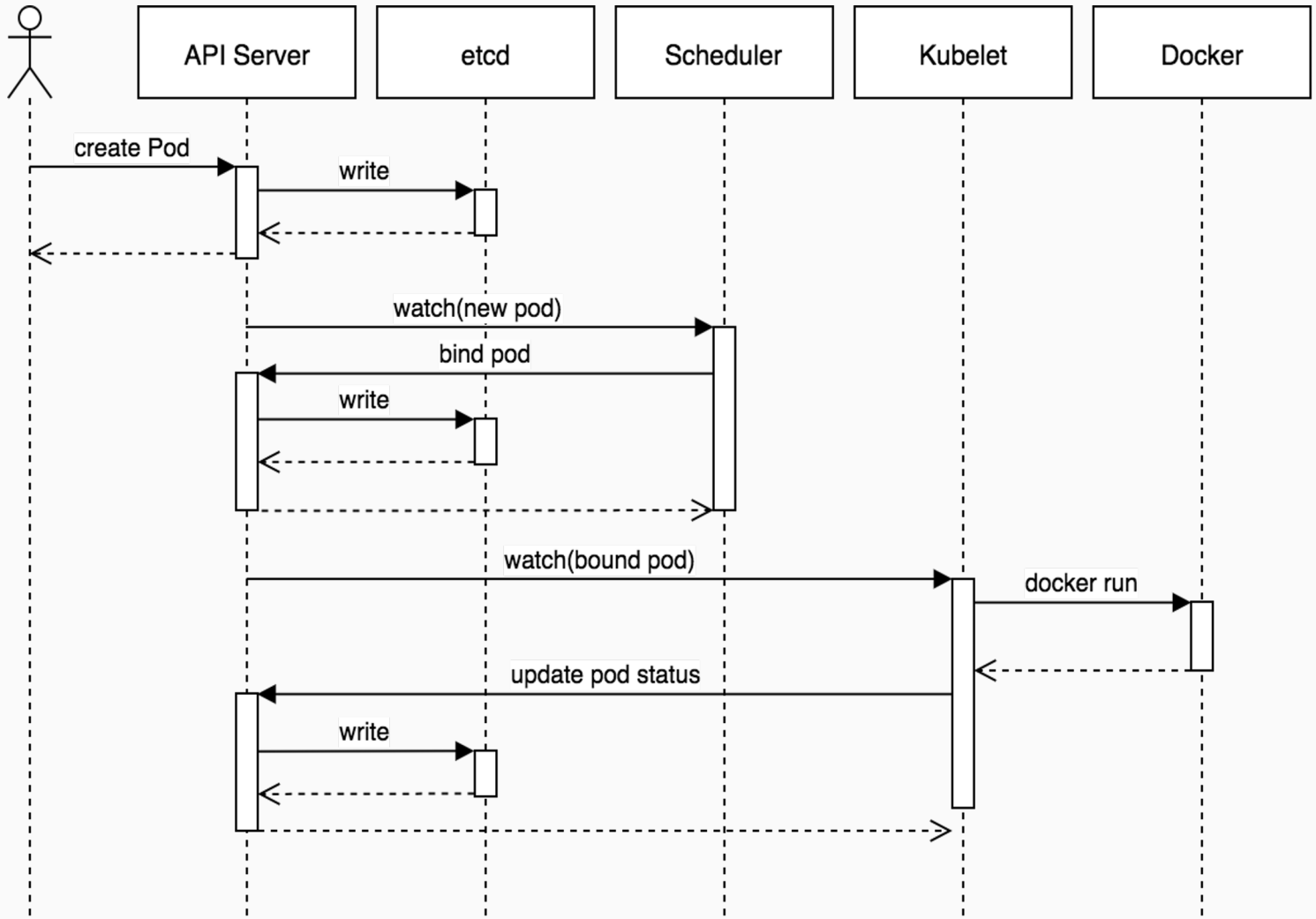
	Resource (abbr.) [API version]	Description
Security	ServiceAccount (sa) [v1]	An account used by apps running in pods
	Role [rbac.authorization.k8s.io/v1]	Defines which actions a subject may perform on which resources (per namespace)
	ClusterRole* [rbac.authorization.k8s.io/v1]	Like Role, but for cluster-level resources or to grant access to resources across all namespaces
	RoleBinding [rbac.authorization.k8s.io/v1]	Defines who can perform the actions defined in a Role or ClusterRole (within a namespace)
	ClusterRoleBinding* [rbac.authorization.k8s.io/v1]	Like RoleBinding, but across all namespaces
	PodSecurityPolicy* (psp) [extensions/v1beta1]	A cluster-level resource that defines which security-sensitive features pods can use
	NetworkPolicy (netpol) [networking.k8s.io/v1]	Isolates the network between pods by specifying which pods can connect to each other
	CertificateSigningRequest* (csr) [certificates.k8s.io/v1beta1]	A request for signing a public key certificate
Ext.	CustomResourceDefinition* (crd) [apiextensions.k8s.io/v1beta1]	Defines a custom resource, allowing users to create instances of the custom resource

Application Dependency on Kubernetes primitives



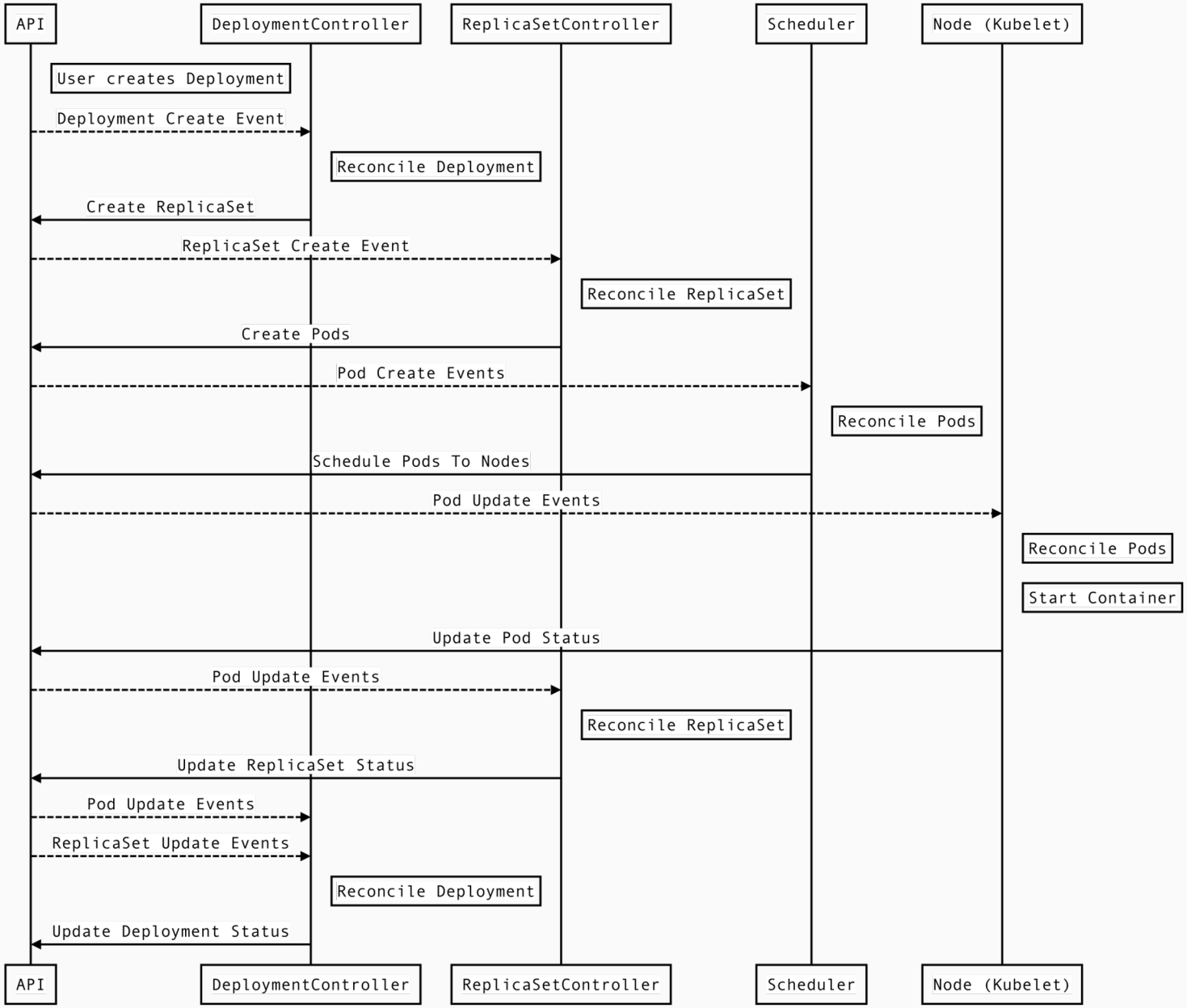
Source: Kubernetes effect by Bilgin Ibryam

How Kubernetes API works: A typical command flow



Source: https://blog.heptio.com/core-kubernetes-jazz-improv-over-orchestration-a7903ea92ca

A more detail Kubernetes Command Flow

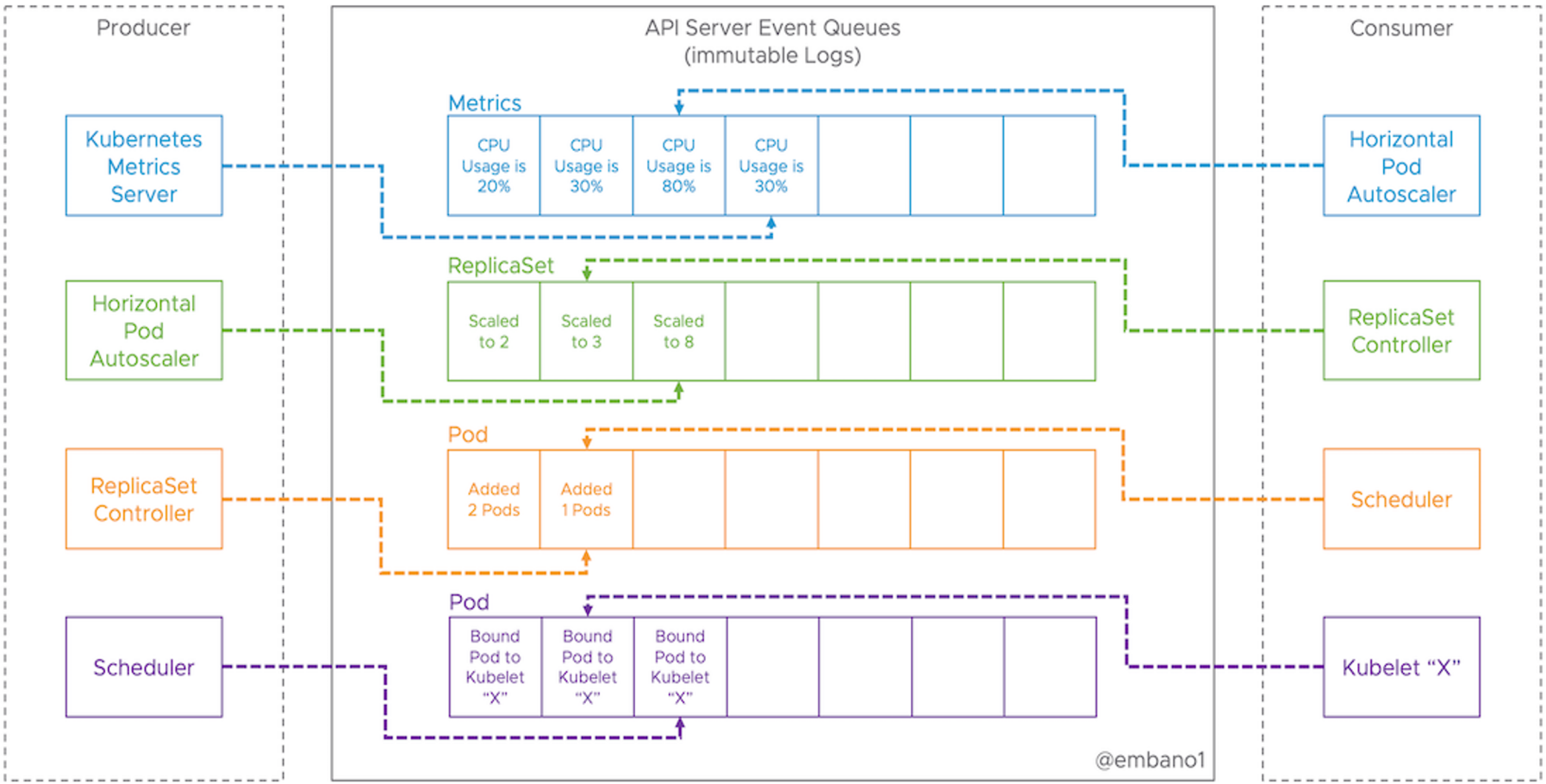


Source: https://medium.com/payscale-tech/imperative-vs-declarative-a-kubernetes-tutorial-4be66c5d8914

The Event-driven Architecture of Kubernetes

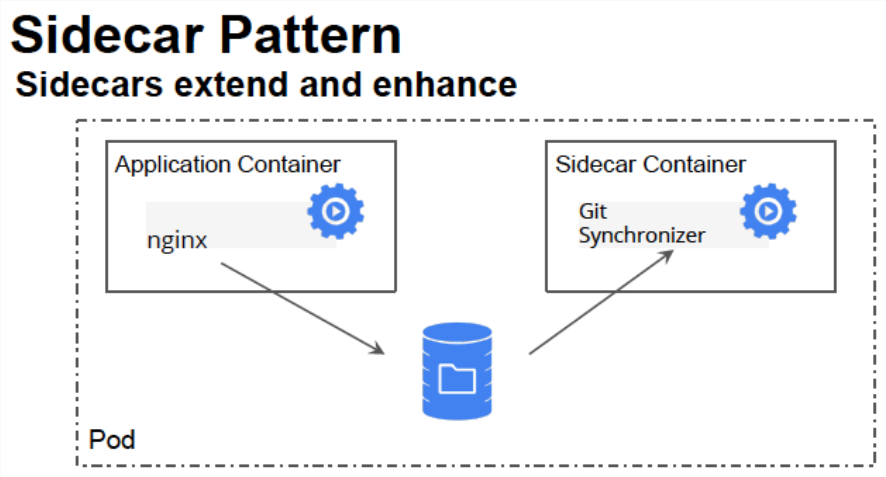
(similar to that of Kafka)

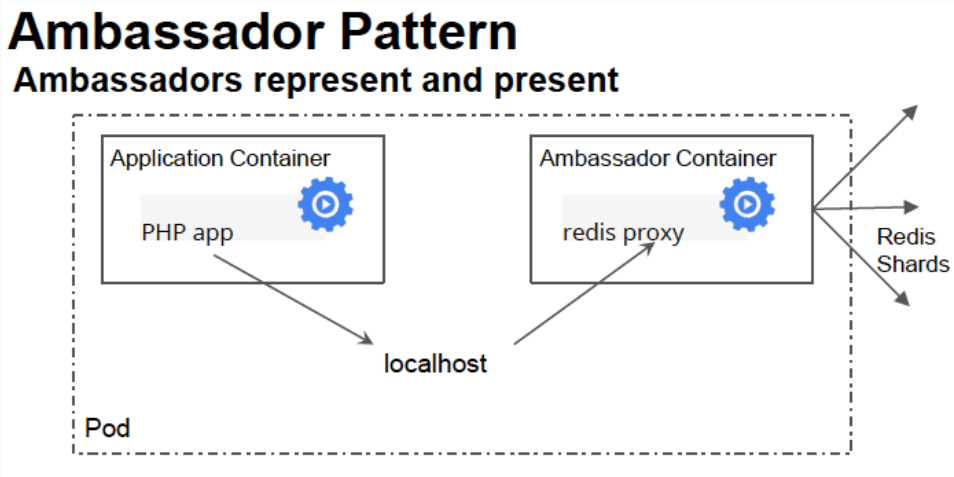
Kubernetes: "Autonomous processes reacting to events from the API server".

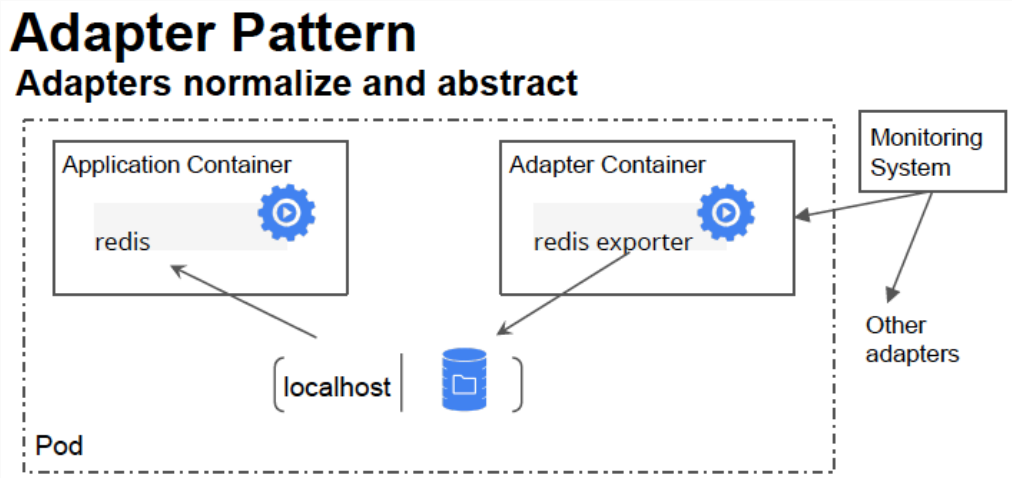


Source: [Events, the DNA of Kubernetes]

Local and Distributed Abstractions/ Patterns

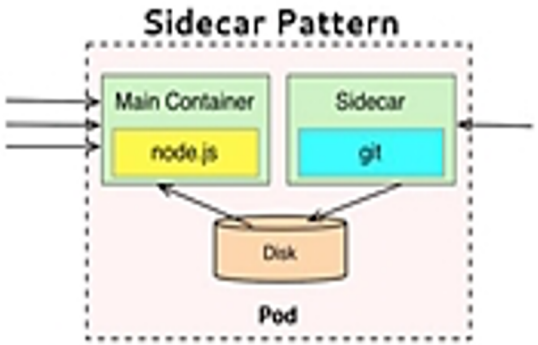


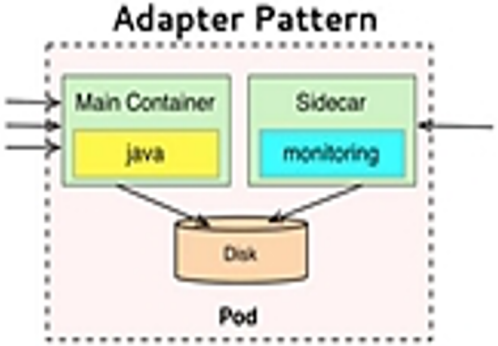


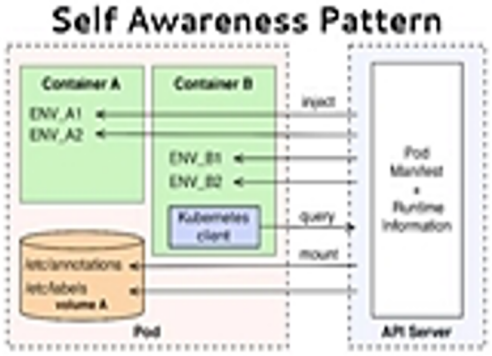


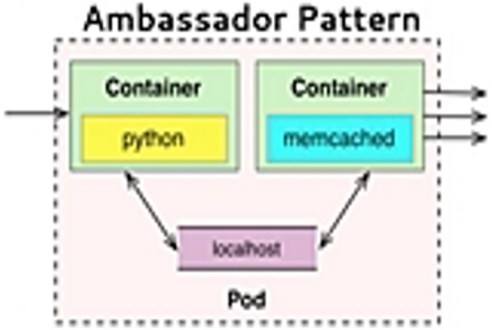
Source: Eric Brewer, ACM SoCC Keynote, 2015

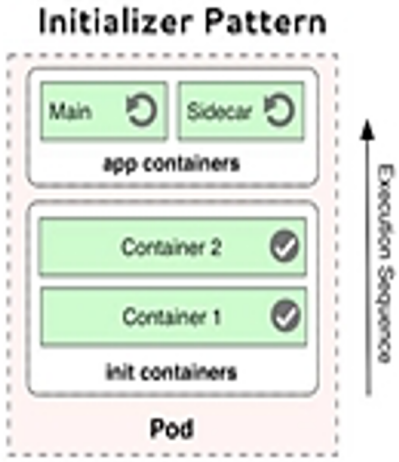
Local and Distributed Abstractions/ Patterns

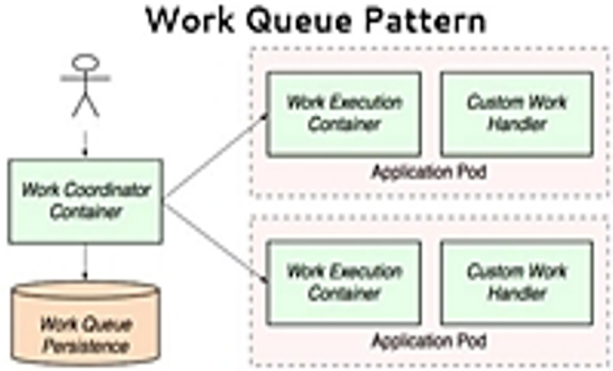












Source: [Kubernetes effect by Bilgin Ibryam]

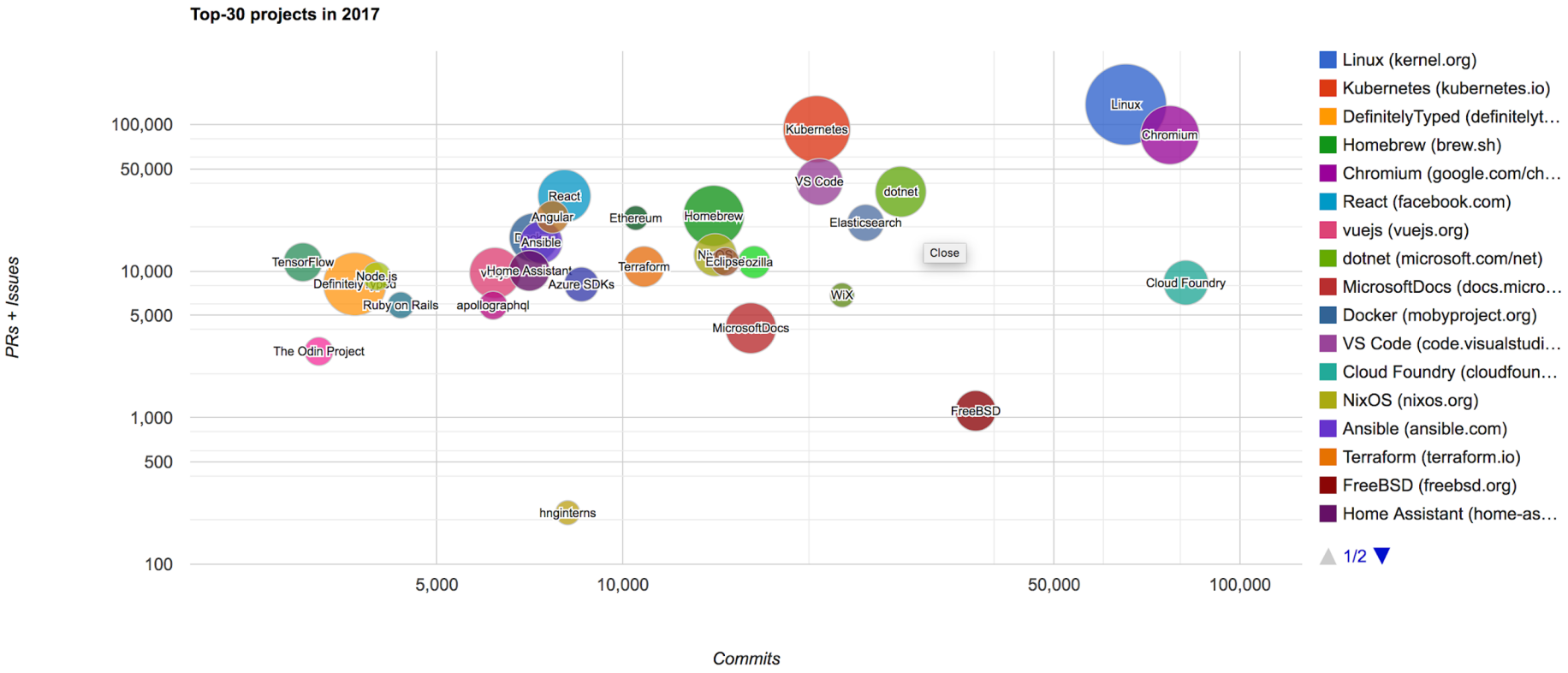
Who “Manages” Kubernetes?

The CNCF is a child entity of the Linux Foundation and operates as a vendor neutral governance group.

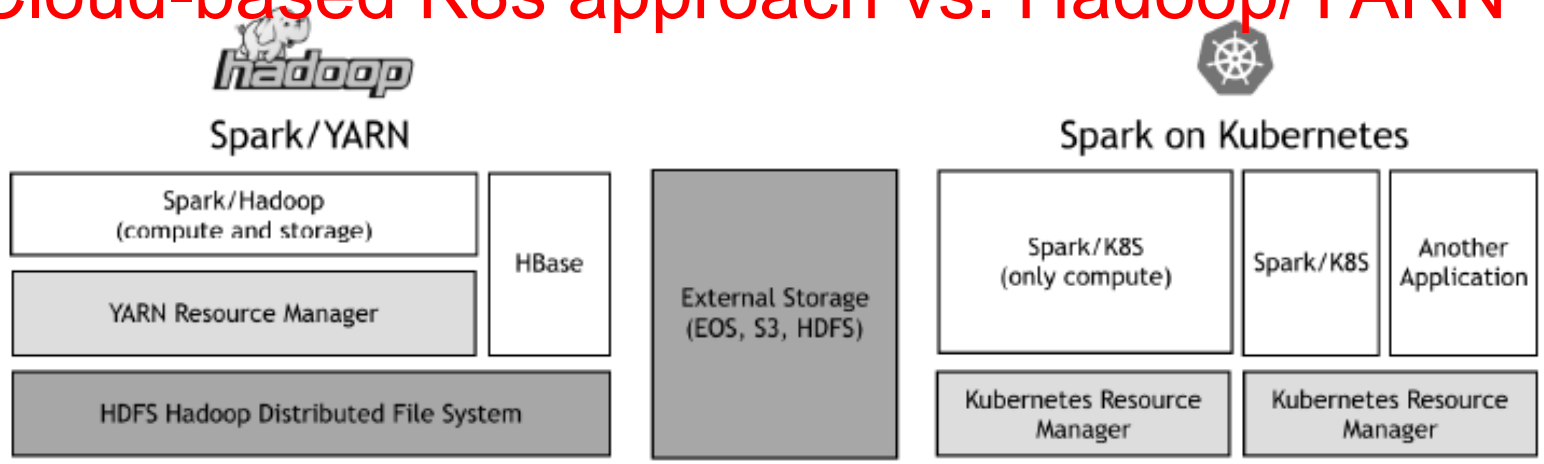
Project Stats for K8s

- Over 55,000 stars on Github
- 2000+ Contributors to K8s Core
- Most discussed Repository by a large margin
- **70,000+** users in Slack Team

Project Stats for K8s



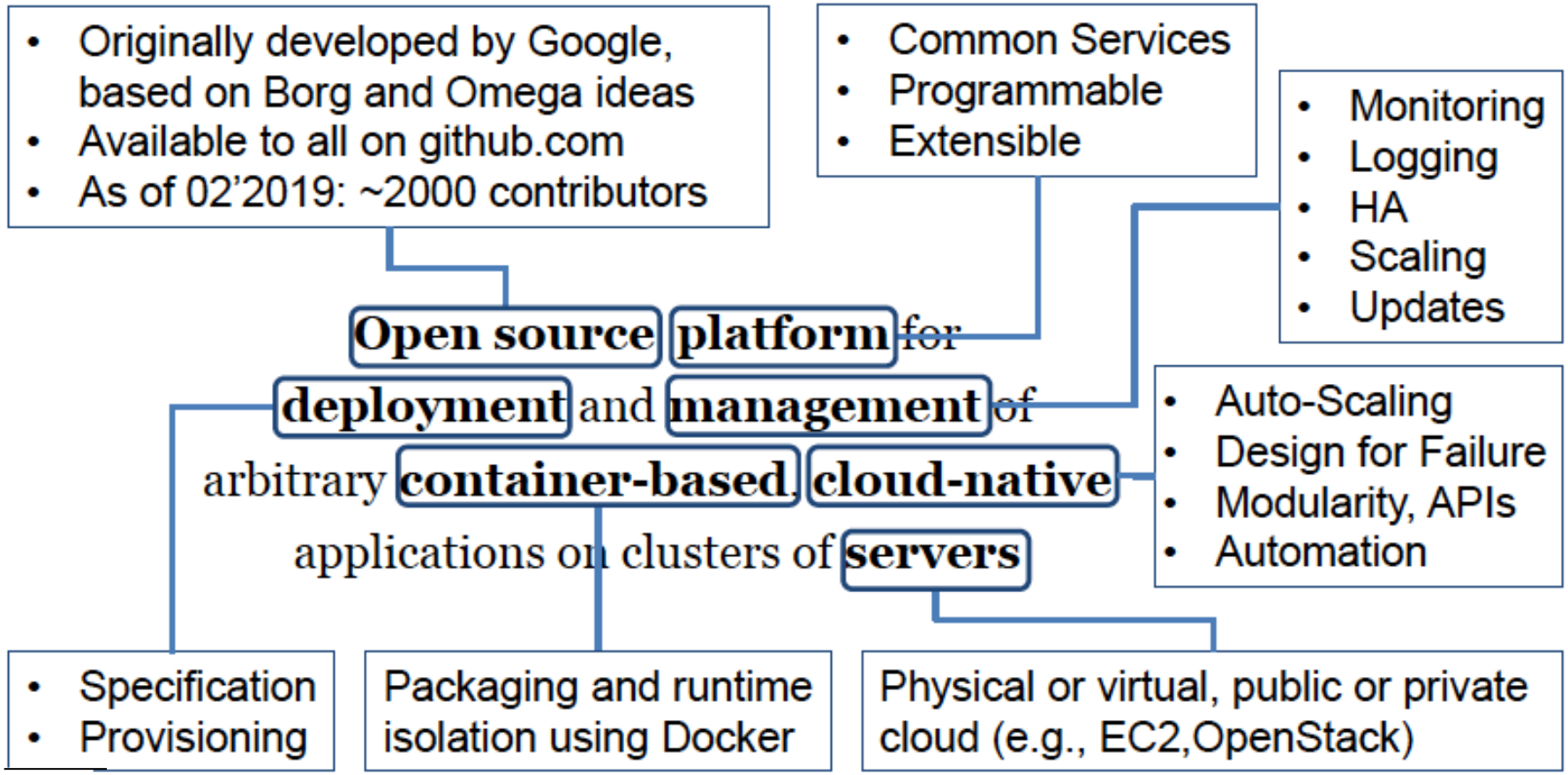
Challenges for Cloud-based K8s approach vs. Hadoop/YARN



- Development of Container-based technologies was geared towards Ephemeral (stateless) Compute-oriented Pods ;
  - Long-term state/ results need to be stored in External Persistent Storage
- For Big Data applications, complex stateful information and data stored on the persistent storage can be large while ephemeral compute nodes may need to be scaled separately
- Trade-offs between Data Locality and Compute Elasticity (Still remember the key idea of "Bringing Computation to the Data" to avoid network-transfer bottleneck ?)
- Deploying Large-scale Persistent (stateful) apps/ services, e.g. Hadoop, over Ephemeral-oriented K8s computing pods remains non-trivial but some solutions are emerging:
  - Cloud Native Storage Solutions include: Container Storage Interface (CSI for K8s), Rook w/ Ceph, Rook w/ Cassandra, Rook w/ CockroachDB etc + Commercial ones, e.g. Trident from NetApp
  - Some Hadoop-over-K8s efforts: e.g. BlueK8 and Kubedirectors, will Hadoop/YARN survive ?
- Steep learning curve for K8s, e.g. for writing/ configuring app package to be deployed over it:
  - YAML, Helm charts, Operators, etc…

Summary of Kubernetes

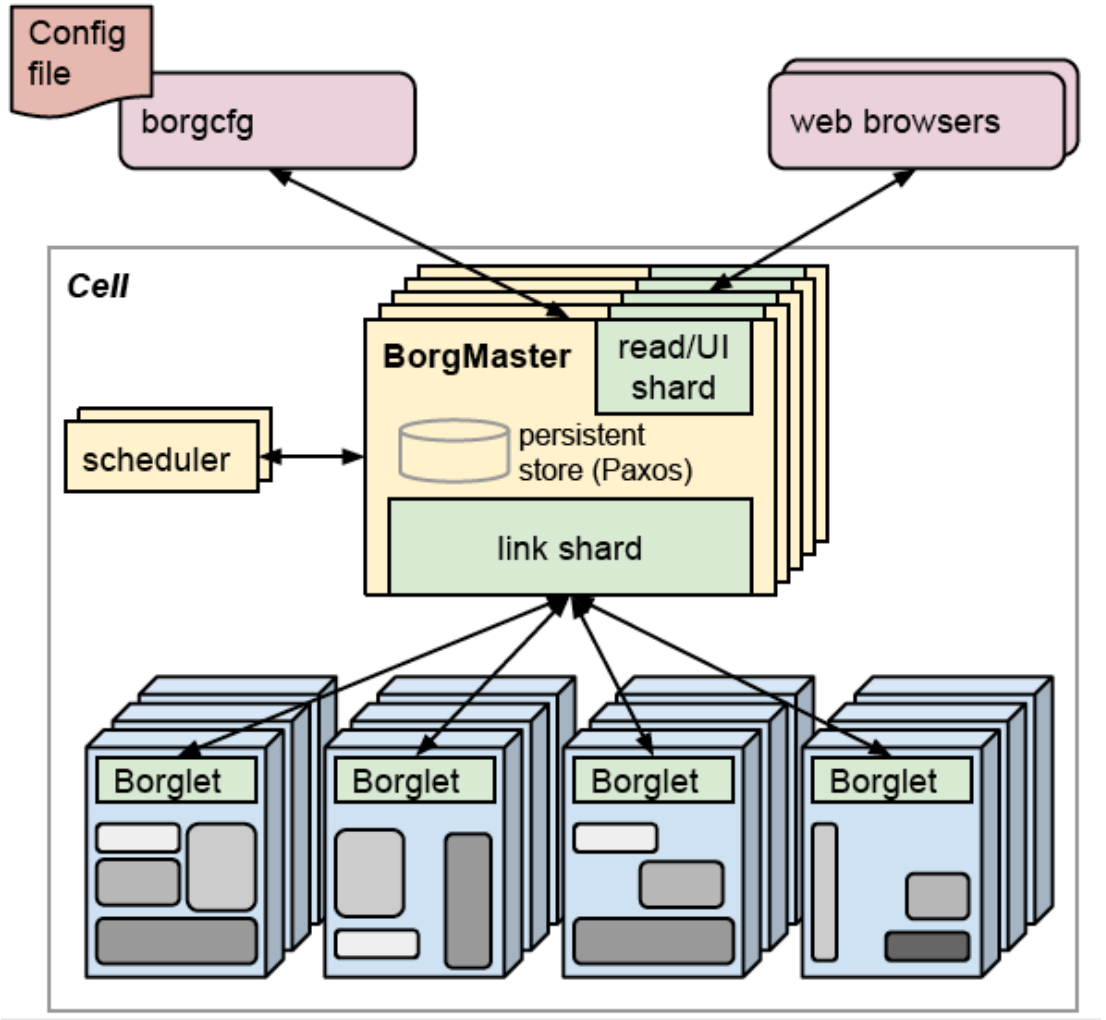
- An Open-source, Cloud-Native Container-based Platform



Backup Slides

More on Borg

High-level Architecture of Google's Borg



A. Verma, L. Pedrosa, "Large-scale cluster management at Google with Borg", Eurosys 2015

Borg Architecture - Borgmaster

- Each cell contains a Borgmaster
- Each Borgmaster consists of 2 processes:
  - Main Borgmaster process
  - Scheduler
- Multiple replicas of each Borgmaster
- Role of (elected leader) Borgmaster:
  - submission of job, termination of any of job's task

Borg Architecture - Borglet

- Local Borg agent on every cell
  - starts/stops/restarts tasks
  - Manages local resources
  - Rolls over debug logs
- Polled by Borgmaster to get machine's current state
- If a Borglet does not respond to several poll messages, it is marked as down
  - Tasks re-distributed
  - If communication is restored, Borgmaster tells Borglet to kill rescheduled tasks

How does Borg work?

- Users submit “jobs”
  - Each “job” contains 1+ “task” that all run the same program/binary
  - Runs inside containers (not VMs as it would cost higher latency)
- Each “job” runs on one “cell”
  - A “cell” is a set of machines that run as one unit

- Two main types of jobs:
  - “Prod” job : long-running server jobs, higher priority
  - “Non-prod” job : quick batch jobs, lower priority

How does Borg work?

- **Allocs:**
  - Reserved set of resources in one machine
  - Can run multiple instances of a task, different tasks from many jobs, or future tasks
- **Priority and quota:**
  - Each job has a priority
  - Preemption disallowed between “prod” jobs.
  - Quota refers to vector of resource quantities for period of time
- **Support for naming and monitoring**

Borg Architecture - Scalability

- Ultimate scalability limit is unknown
  - Single Borgmaster can manage thousands of borglets
  - Rates above 10,000 tasks per minute
  - Busy Borgmaster uses 10-14 CPU cores and 50GiB RAM

Isolation

- Security:
  - Linux *chroot* command used for process isolation
  - Standard sandboxing techniques used for running external software
- Performance:
  - Borg makes explicit distinction between LS (latency-intensive) tasks and batch tasks. Helps for priority-based preemption
  - Borg uses notion of compressible resources (CPU cycles, disk I/O bandwidth) and non-compressible resources (RAM, disk space)

Borg Architecture - Scheduling

- Borgmaster adds new jobs to a pending queue after recording it in the Paxos store
- A scheduler (primarily operates on tasks) scans and assigns tasks to machines
  - Feasibility checking
  - Scoring
- E-PVM vs “best-fit”
  - E-PVM leaves headroom for load-spikes but has increased fragmentation
  - Best-fit fills machines as tightly as possible, but hurts “bursty loads”
- Current model is a hybrid of both
  - Borg will kill lower priority tasks until it finds room for an assigned task

Techniques used by Borg for Scalability:

- Score caching
  - Feasibility and scoring is expensive, scores are cached until properties of the machine or task change
- Equivalence class
  - A group of tasks with identical requirements
  - Stems from tasks within a job having identical requirements and constraints
  - Easier than determining feasibility for every pending task and scoring every machine
- Relaxed randomization
  - Scheduler examines machines in a random order until it has found enough feasible machines to score, and selects the best from this set
  - Speeds up assignments of tasks to machines

Borg - Achieving Availability

- To mitigate inevitable failures, Borg will:
  - Automatically reschedule evicted tasks
  - Reduce correlated failures by distributing across failure domains
  - Limits downtime due to maintenance
  - Use “declarative desired-state representations and idem-potent mutating operations” to ease resubmission of forgotten requests
  - Avoid task to machine pairings that cause crashes
  - Use a logsaver to recover critical data written to a local disk
- Achieve 99.99% availability in practice

Utilization - Main Goal of Borg

- Efficient utilization is very important for Google:
  - A few percentage improvement can save millions of dollars!
- Cell Sharing
- Large Cells
- Fine-grained Resource Requests
- Resource Reclamation

Borg/Kubernetes Comparisons:

- Borg is a predecessor to Kubernetes
- Borg groups work by 'job'; Kubernetes adds 'labels' for greater flexibility.
- Kubernetes allows for Docker and other containers, while Borg only seems to allow LMCTFY
- Single IP per machine in Borg complicates things
  - Because of Linux namespaces, VMs, IPv6, and software-defined networking, Kubernetes assigns every “pod” and service its own IP address
  - Allows developers to choose ports and removes the infrastructure complexity of managing ports
- Borg is not open source or available for use outside of Google unlike Kubernetes
  - Both work on bare metal, but Kubernetes can work on various cloud hosting providers, “such as Google Compute Engine.”

How does the Borgmaster handle load spikes while minimizing fragmentations?

- Hybrid of E-PVM (worst-fit) and best-fit model
  - E-PVM leaves headroom for large spikes, has large fragmentation
  - Best-fit model fills machines as tightly as possible
    - Pre-empts by killing lower priority tasks and add it back to pending queue
  - Large cells used to decrease fragmentation

Techniques Borg uses for managing utilization

- Cell-sharing: sharing prod and non-prod tasks
  - Resource reclaiming
  - Not sharing prod and non-prod work would increase machine needs by 20-30%
- Large cells: to allow large computations and decrease fragmentation
  - splitting up jobs and distributing them requires significantly more machines
- Fine-grained resource requests
  - fixed size containers/VMs not ideal
  - instead there are “buckets” of CPU/memory requirements
- Resource reclamation: jobs specify limits
  - Borg can kill tasks that use more RAM or disk space than requested
  - Throttle CPU usage
  - Prioritize prod tasks over non-prod

Why is it important to have isolation, and how does Borg implement it?

- To protect an app from Noisy, Nosy and Messy neighbors
- Sharing machines between applications increases utilization, but isolation is needed to prevent tasks from interfering
  - Security: rogue tasks can affect other tasks, and information should not be visible between tasks
  - Performance:
    - Utilization can be decreased by users inflating resource requests to prevent interference
    - Again, rogue tasks can affect your task
- Security: Linux chroot jail is the primary security isolation mechanism
- Performance: Linux cgroup-based container
  - Also appclass is used to help with overload and overcommitment
  - High priority LS (latency-sensitive) tasks