Cloud Resource Management

Introduction

1. Cloud is multi-tenant by its very nature with QoS guarantees for different applications and different programming frameworks
    - Computational resources are CPUs, memory footprint, network bandwidth and latency and must be accounted for when scheduling resources
    - Look at state-of-the-art for managing cloud resources

Resource Management for the Cloud

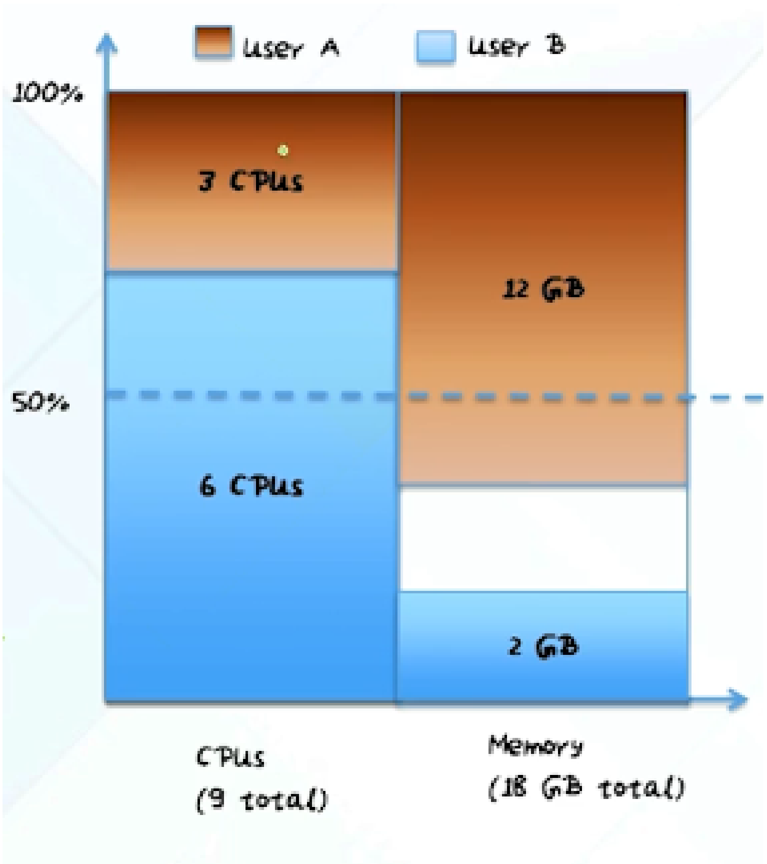
1. Context and terminologies
    - Resources in the context of cloud applications
        - CPU
        - Memory footprint and effective memory access time
        - Storage (bandwidth, latency)
        - Network (bandwidth, latency)
    - Resource utilization
        - Percent time a resource is actively used
    - An application may use all or most of the resources
    - Begs the questions
        - Which resoure's utilization is more important?
        - Should we optimize for one or multiple resources' utilization?
    - Basics
        - CPU scheduling in traditional OS
            * Focus on CPU utilization
            * FCFS (first come first served), SJF (shortest job first), Round-robin, SRTF (shortest remaining time first), priority queues, multi-level priority queues…
        - Fairness
            * User's perception of resource allocation policy
            * e.g., round-robin gives a feeling of fairness if all processes are created equal; Each process gets 1/N processor resource, where N = number of processes
        - What if all processes are not created equal?
            * "Fair sharing" takes into account priorities; you get what you pay for
    - Variants of fair sharing
        - Equal share
        - Max-min fairness
        - Weight max-min fairness
    - How do we extend "fair sharing" to the cloud?
        - Need to consider all the resources, not just CPU
    - Computations running in data centers are not all uniform
        - Data intensive workloads (health informatics)
        - Compute intensive workloads (ML algorithms)
        - Network intensive workloads (Netflix)

Fair Share Schedulers

1. Hadoop
    - Uses max-min scheduling
        - K-slots per machine (CPU + memory)
            * A slot is a fraction of the machine's resources
    - A job may consist of a number of tasks
    - Assign one task per slot
    - Apply max-min fairness in mapping tasks to slots

- Could result in under-utilization of slot resources or over-utilization, e.g. memory thrashing
  - If a task needs 128 MB, it will lead to thrashing in the above example
  - If it only needs 32 MB then half the allocation is wasted

2. Dominant Resource Fairness (DRF)
   - Takes a holistic view of a machine's resources
     - Total resource of a machine: 10 CPUs and 4 GB
     - Task needs: 2 CPUs, 1 GB
       * Dominant resource of task 1 is memory
       * If task 1 is allocated a slot on this machine its "dominant resource share" is 25%
   - DRF uses max-min fairness to dominant shares
     - System: 9 CPUs + 18 GB
     - A: each task 1 CPU + 4 GB (dominant: memory)
     - B: each task 3 CPU + 1 GB (dominant: CPU)
     - Equalize allocation for A and B on dominant shares
       * Each gets 2/3 of the dominant share



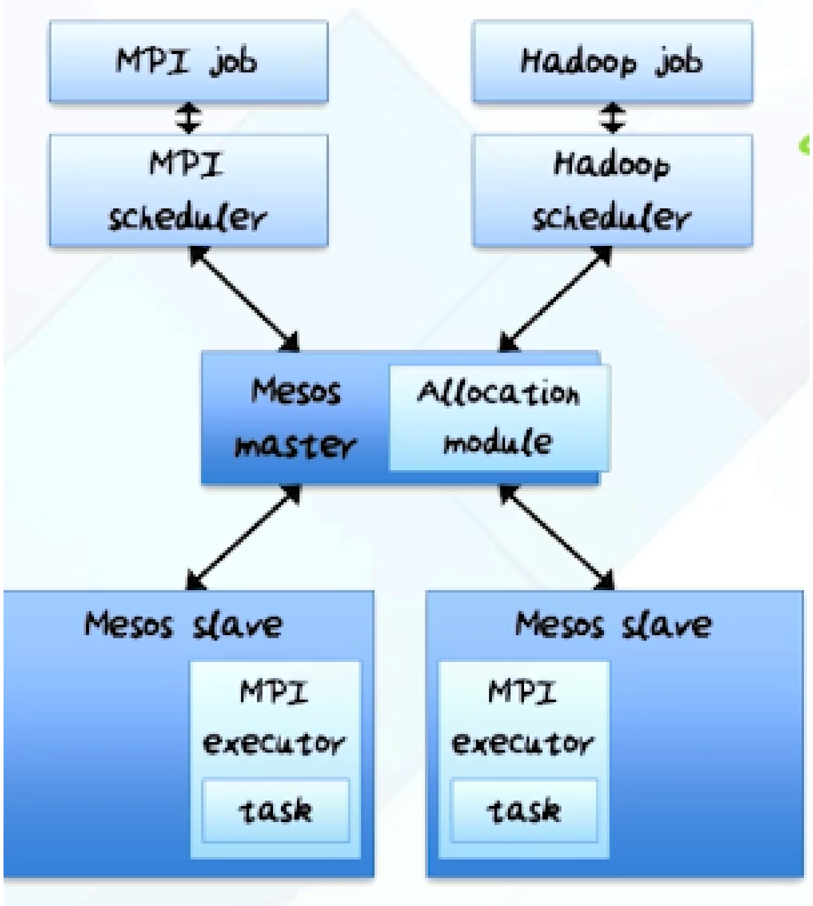
Dominant Resource Fairness

Mesos

1. Challenge for resource sharing in the cloud
   - Applications use a variety of frameworks
     - Map-reduce Dryad, Spark, Pregel
   - Apps written in different frameworks need to share the data center resources simultaneously
     - 1000s of nodes, hundreds of "jobs", each with millions of (small duration) tasks

- Need to maximize utilization while meeting Qos needs of the apps

2. Traditional approach to resource sharing
   - Static partitioning (cluster with 9 machines)
     - Each framework gets 3 machines
3. Optimal approach
   - Global schedule
     - Inputs: Requirements of frameworks, resource availability, policies (e.g., fair sharing)
     - Output: Global schedule for all the tasks
     - Not practical: Too complex
       * Not scalable to the size of cluster and number of tasks
       * Difficult to implement in the presence of failures
   - Mesos goal
     - Dynamic partitioning leading to high utilization
4. Mesos approach: Fine-grained resource sharing
   - A thin layer (a la micro-kernel) allocation module between the scheduler of the frameworks and the physical resources

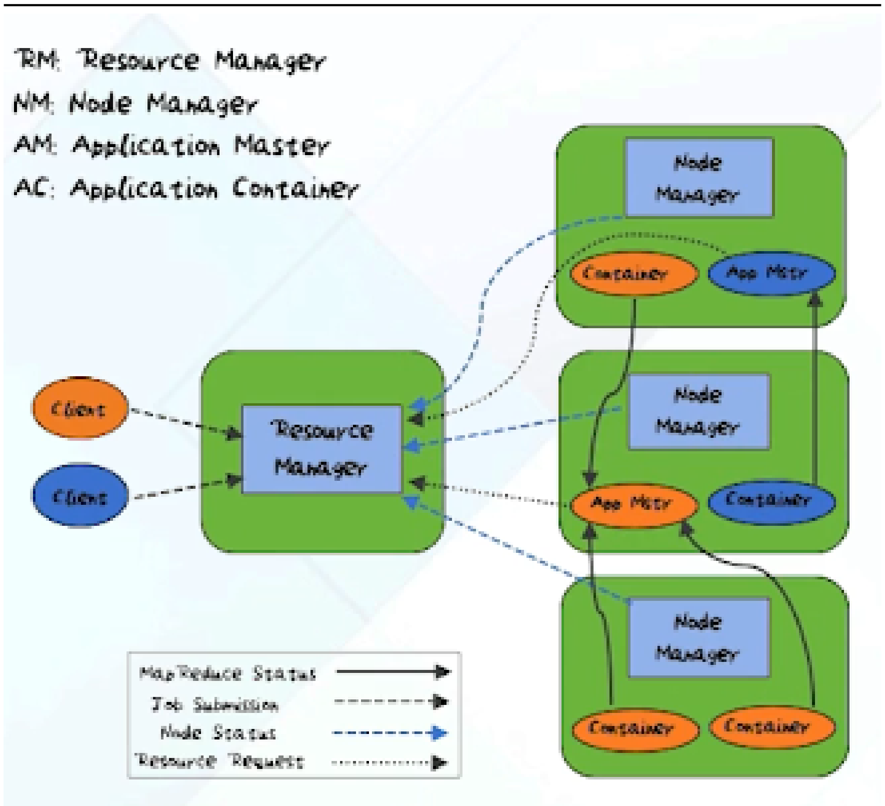


Mesos

5. Mesos: Resource offers
   - Make offers to the frameworks
   - Framework assigns tasks based on the offer
   - Framework can reject the offer if it does not meet its needs
   - Solution may not be optimal but resource management decisions can be at a fine grain and quick

Hadoop YARN

1. YARN: Yet Another Resource Navigator
    - Similar goal to Mesos
        - Resource sharing of the cluster for multiple frameworks
    - Mesos if "offer" based from allocation module to the application frameworks
    - YARN is "request" based from the frameworks to the resource manager
2. YARN background
    - Traditional Hadoop (open source implementation of map-reduce)
        - Job-tracker/Task-tracker organization
        - Poor cluster utilization
            * Distinct map/reduce slots
3. YARN details
    - Client submits app to resource manager (RM)
    - RM launches application master (AM) in a container
    - AM registers with RM at boot-up, requests application containers (ACs) to RM
    - RM allocates ACs in concert with the node manager (NM)
    - AM contacts the NM to launch its ACs
    - Upon application completion, AM shuts down
    - YARN can implement different scheduling policies
        - FCFS, fair, capacity

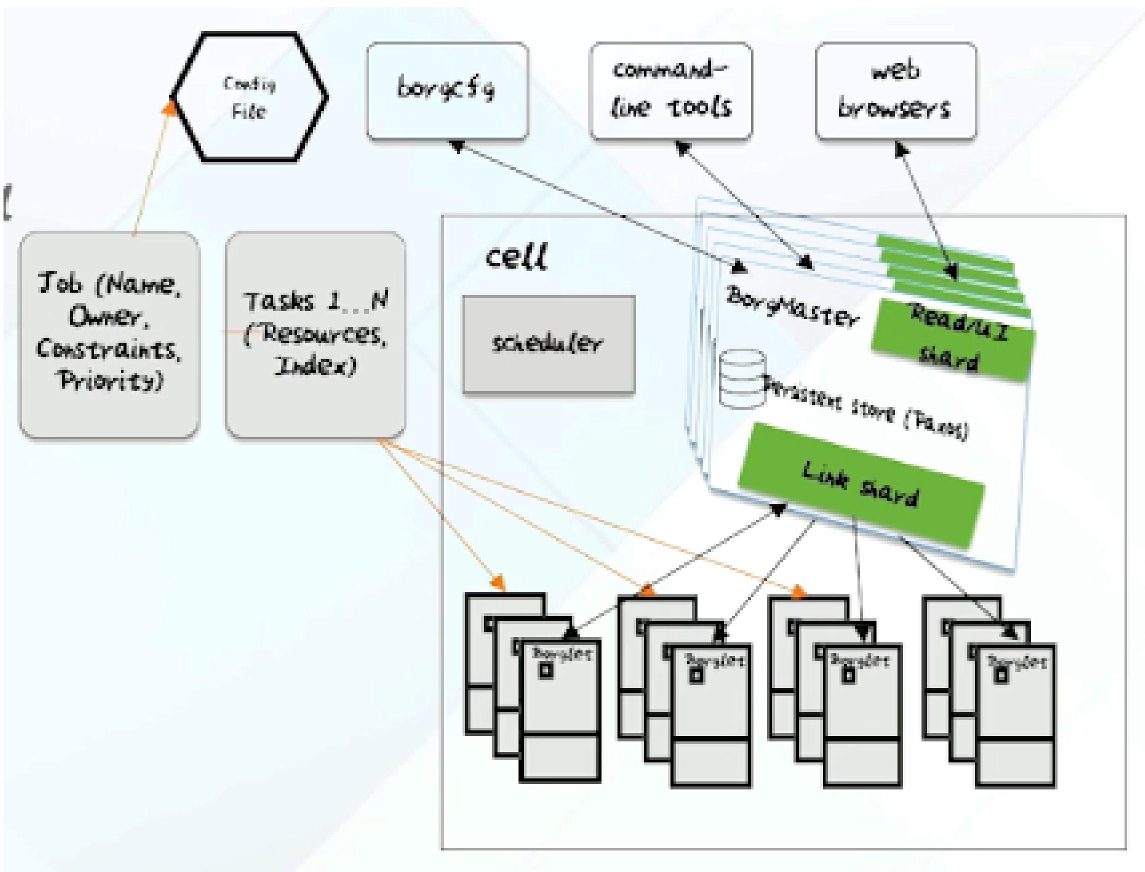


YARN

Google Borg Resource Manager

1. Google's Resource Manager
    - Similar in spirit to Yahoo's YARN and Mesos

        - Provide data center resources for a variety of applications, written in a variety of frameworks, running simultaneously
2. Borg terminologies
    - Cluster: A set of machines in a building
    - Site: Collections of buildings
    - Cell: A subset of machines in a single cluster
        - Unit of macro resource management
3. Borg priority bands
    - Production jobs
        - Higher priority jobs (e.g., long-running server jobs)
    - Non-production jobs
        - Most batch jobs (e.g., web crawler)
    - Production jobs higher priority
    - Non-overlapping "priority bands"
        - Monitoring, production, batch, best-effort
4. Borg architecture and scheduling
    - BorgMaster: Per cell resource manager
    - Borglet: Agent process per machine in cell
    - Scheduler
        - High-low priority jobs
        - Round robin within a band
        - Consults BorgMaster for resources
        - BorgMaster
            - Feasibility checking
            - Scoring
        - Tasks
            - Run in containers



Google Borg

5. Kubernetes
    - An open-source resource manager derived from Borg
        - Runs on Google cloud
        - Uses Docker containers
            - Resource isolation
            - Execution isolation

Mercury

1. Mercury background
    - Framework for integrated centralized and distributed scheduling
    - Centralized schedulers
        - Mesos, YARN, Borg
        - Pro: Simplifies cluster management via centralization
        - Con: Single choke point -> Scalability and latency concerns
    - Distributed schedulers
        - Independent scheduling decision making by jobs
            - Allows jobs to directly place tasks at worker nodes -> less latency
        - Con
            - Not globally optimal
2. Mercury insight
    - Combine virtues of both centralized and distributed
        - Resource guarantees of “centralized”
        - Latency for scheduling decisions of “distributed”

    - Application choice
        - Trade performance guarantees of allocation latency
3. Mercury architecture
    - Central scheduler
        - Policies/guarantees
        - Slower decision making
    - Distributed scheduler
        - Fast/low-latency decisions
    - AM specifies resource type
        - Guaranteed containers
        - Queueable containers

Mercury

Conclusion

1. Cloud resource management technologies are not unique from what occurs in distributed systems
    - Scale at which these algorithms must operate and QoS requirements make building such software a non-trivial engineering exercise