Resource Scheduling Architectural Evolution at Scale and Distributed Scheduler Load Simulator

Renyu Yang

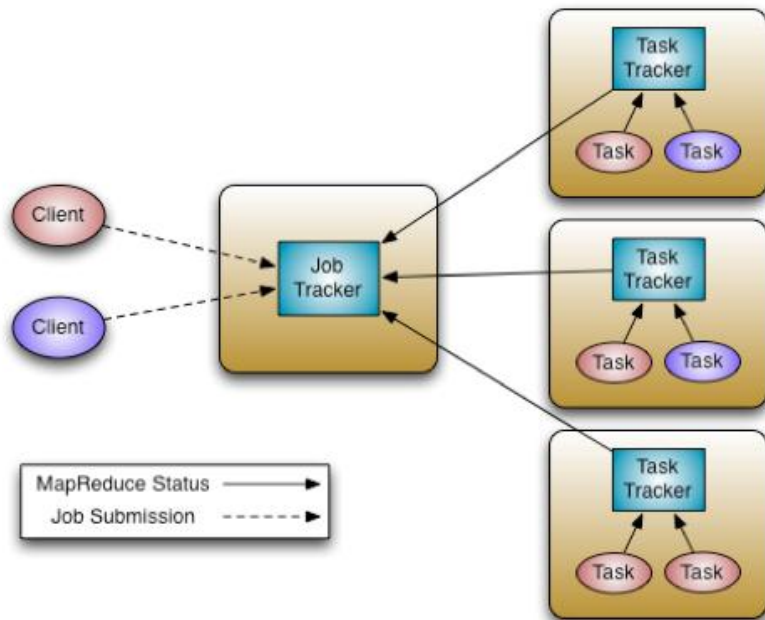# Resource Scheduling Problems

Job

Tasks

Machines
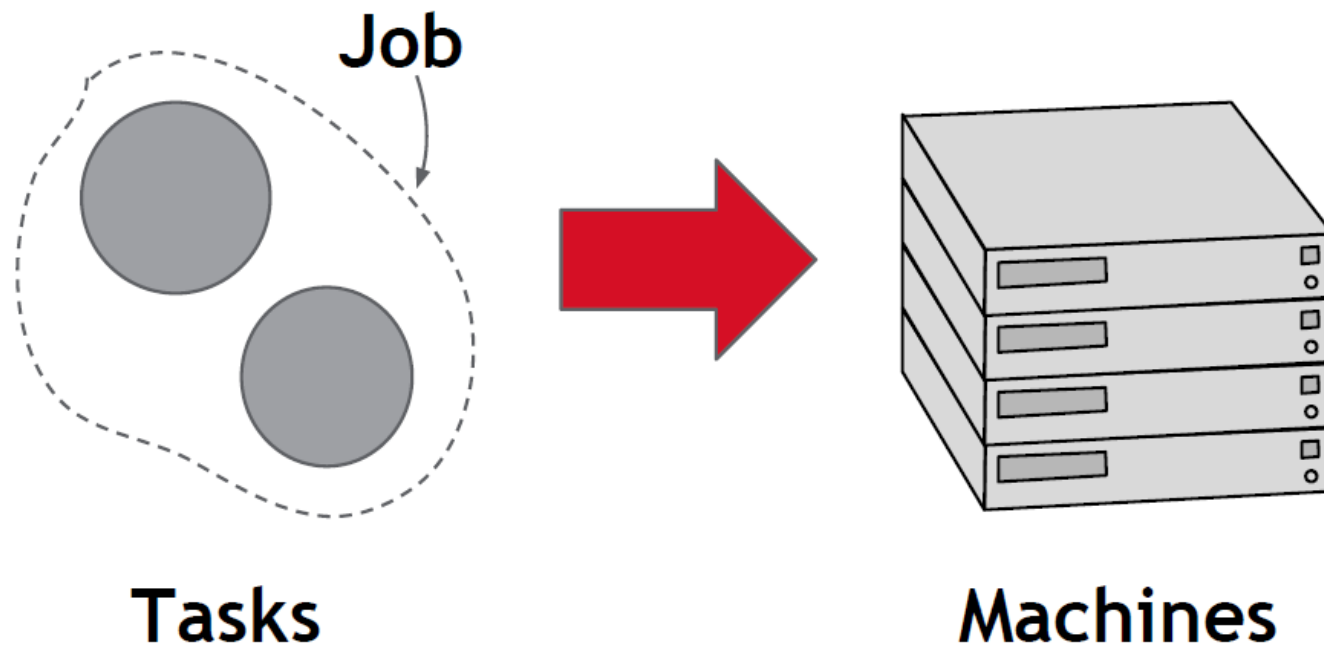
Challenges at Scale

- Perspectives：
  - Performance
  - Scalability
  - Fault-tolerance
  - Resource Utilization

- Case studies:
  - System Architectural Evolution
  - Design of scheduler load simulator based on different resource management models.

Data Processing System Overview

- **1st Generation(G1)**:
  - Hadoop Map Reduce programming paradigm
- **2nd Generation(G2)**:
  - Mesos
  - Yarn: short for "Yet Another Resource Negotiator"
  - Fuxi: Alibaba Cloud Inc.
- **3rd Generation**？
  - Omega from Google

# G1: Hadoop MR classic

- JobTracker
  - Manages cluster resources
  - Task scheduling
    - how to divide tasks
    - which node, how much resource for execution

- TaskTracker
  - Per-machine agent and daemon
  - Manage each execution individual task

5

Limitations

- Problems with large scale
  - > 4000 nodes
  - > 40k concurrent tasks
- Problems with resource utilization
- Only support one one type of computing paradigm (Slots only for Map or Reduce)
- Overloaded Job Tracker, Single point of failure!
- Restart is very tricky due to complex states → weak failover mechanism

Increasing Requirement
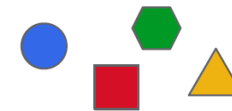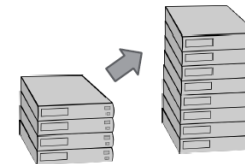
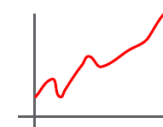• Rapid innovation in cluster computing frameworks
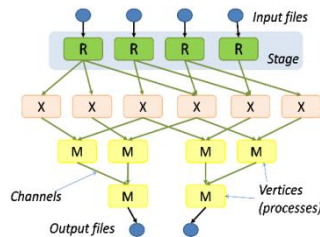
Trend observed !

Dryad

Pregel

Percolator

Diverse workloads
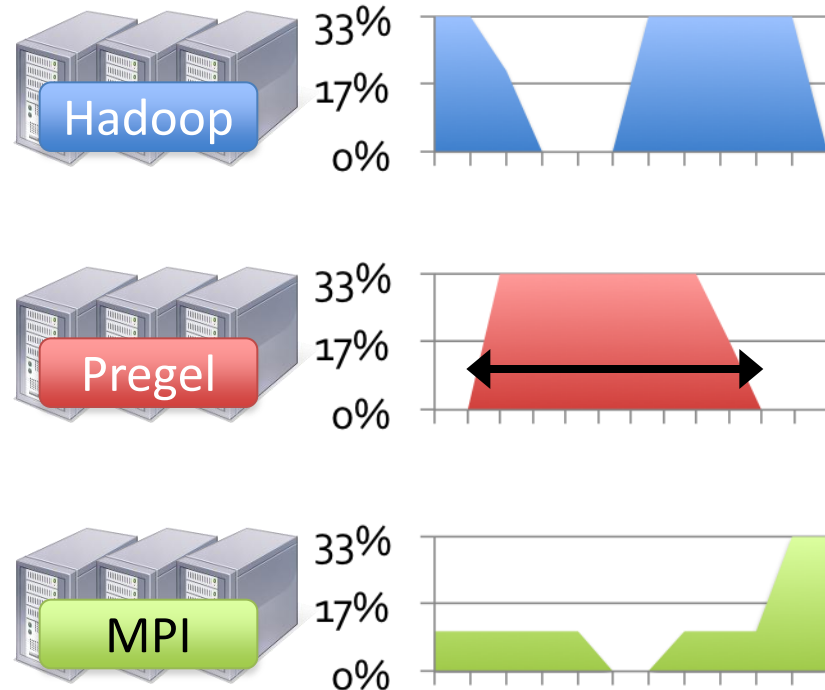
Increasing cluster sizes

Growing job arrival rates

Users would like to run both existing and new application frameworks on the same physical clusters and at the same time.
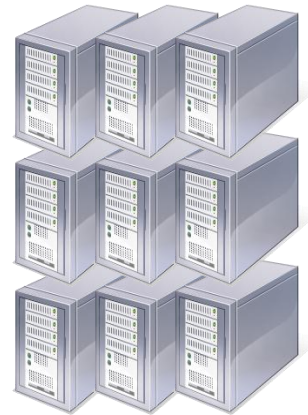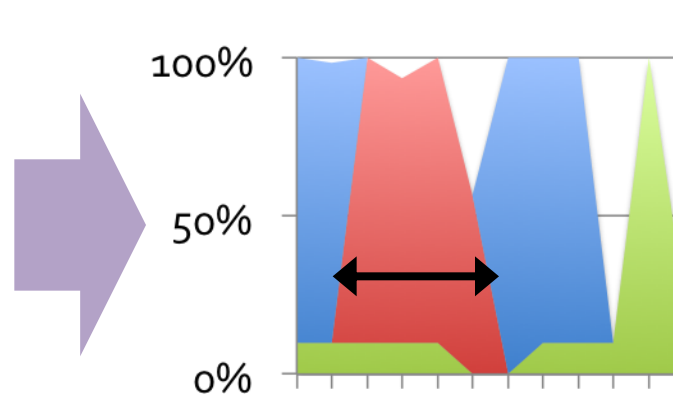
Motivation

No single framework optimal for all applications

Today: static partitioning

dynamic sharing

Shared cluster
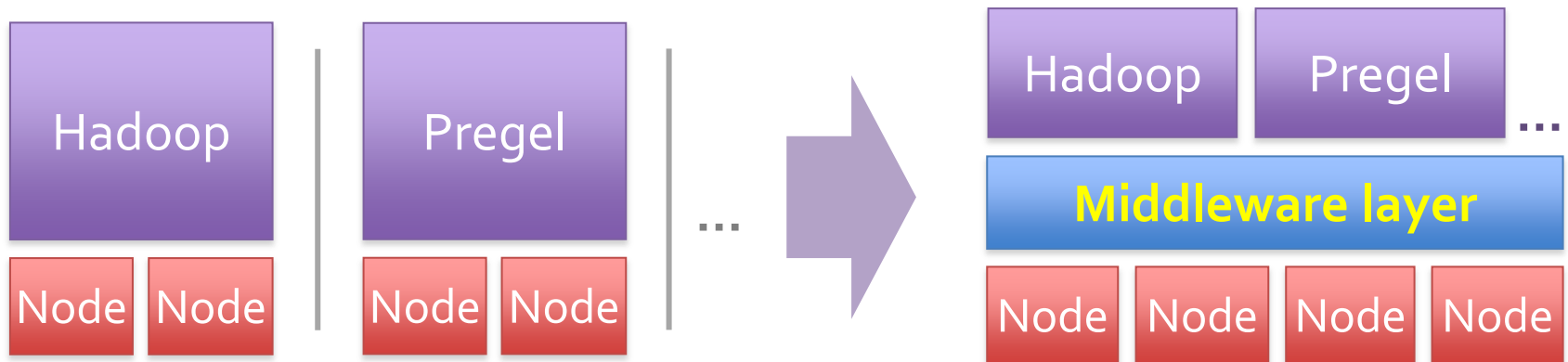
We want to run multiple frameworks in a single cluster
...to *maximize utilization*
...to *share data* between frameworks

2nd Generation(G2) Solutions

- We need a common resource sharing layer over which diverse frameworks can run.
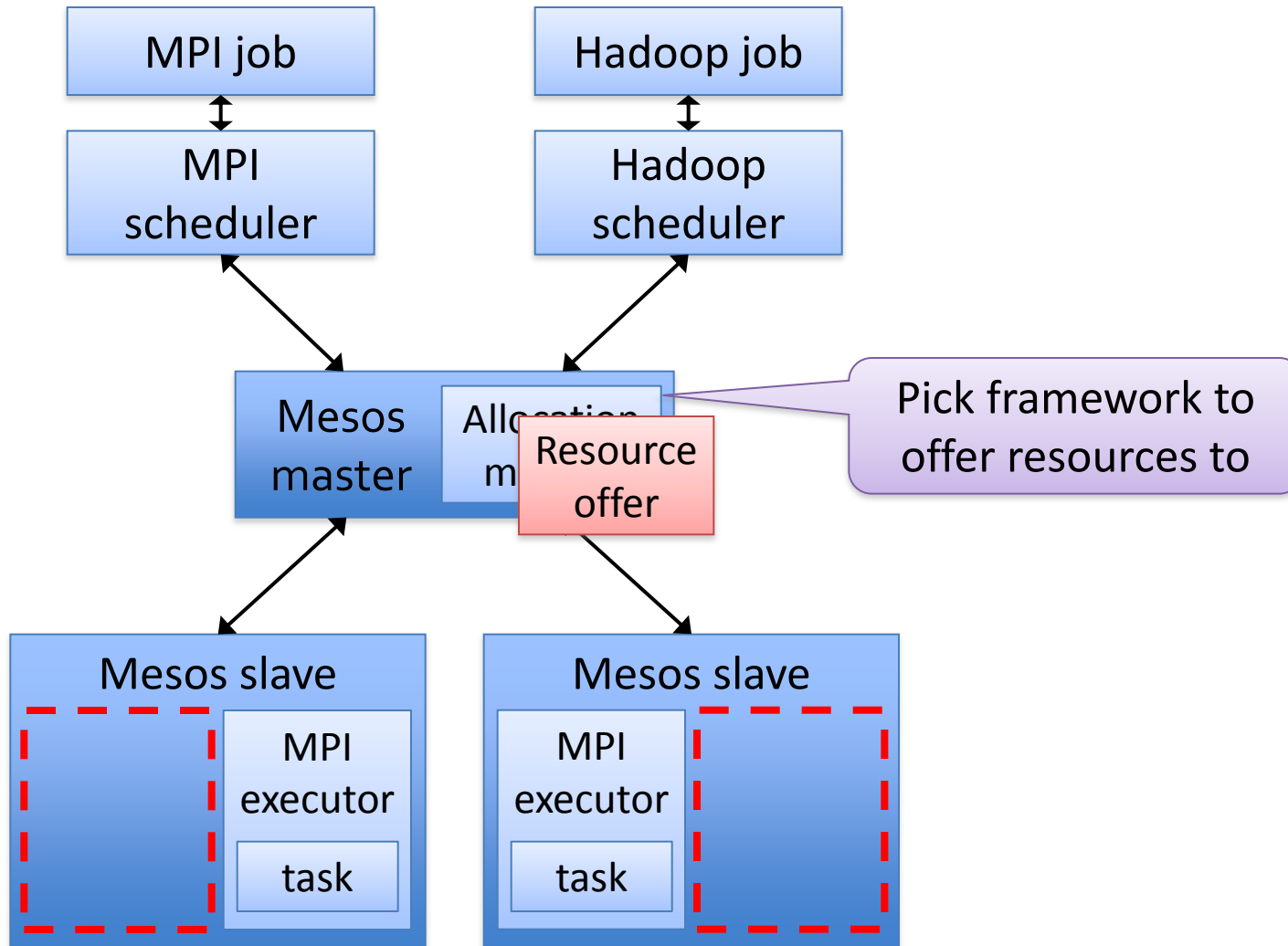

- Goals
  - **High utilization of resources**
  - **Support diverse frameworks**
  - **Better scalability to 10,000's of nodes**
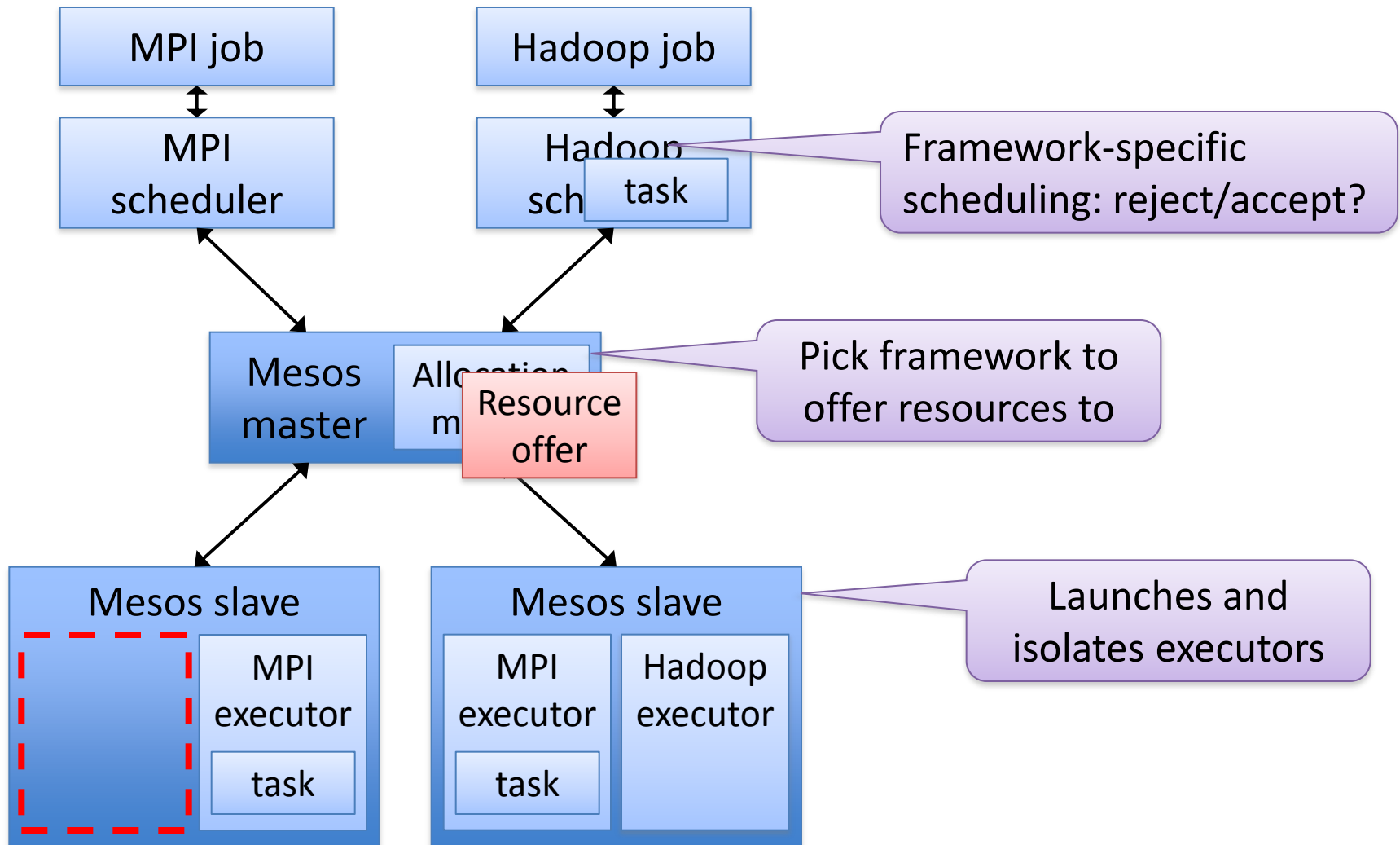 - **Improved reliability in face of failures**

Mesos from Berkeley

- Philosophy: **"offer and accept"** for resource

- **Resource allocation module in Mesos** decides how many resources should be offered to each application framework, based on an organizational policy such as fair sharing (e.g., DRF)

- Frameworks decide which resources to accept and which tasks to run on them.

Mesos Architecture

MPI job

Hadoop job

MPI scheduler

Hadoop scheduler

Mesos master

Allocation m...

Resource offer

Pick framework to offer resources to

Mesos slave

MPI executor

task

Mesos slave

MPI executor

task

Mesos Architecture

MPI job

Hadoop job

MPI scheduler

Hadoop scheduler

task

Framework-specific scheduling: reject/accept?

Mesos master

Allocation module

Resource offer

Pick framework to offer resources to

Mesos slave

MPI executor

task

Mesos slave

MPI executor

task

Hadoop executor

Launches and isolates executors

Limitations

- Passive offer-based policy
  - Only accept or reject what is on offer but cannot specify any request
  - Severed order of each framework depends on the offering order
  - Risk of long-time resource starving
 - Can not support resource preemption

G2++: **Next Generation MRv2**

- De-coupling *JobTracker* into：
  - Resource Management
  - Scheduling / Monitoring

- Following a request-based and active approach that improves scalability, resource utilization, fault tolerance.

- Providing slots for jobs other than Map / Reduce

YARN

❖ *JobTracker* is de-coupled into

  ❖ Global Resource Manager - Cluster resource management

  ❖ Application Master - Job scheduling and monitoring (one per application). The Application Master negotiates resource containers from the Scheduler, tracking their status and monitoring for progress. Application Master itself runs as a normal container.

❖ *TaskTracker* is simplified into

  ❖ NodeManager (NM) - A new per-node slave that is responsible for launching the applications' containers, monitoring their resource usage (CPU, memory, disk, network, etc) and reporting back to the Resource Manager.

❖ YARN maintains compatibility with existing MapReduce applications and users.

Yarn's Architecture and Workflow
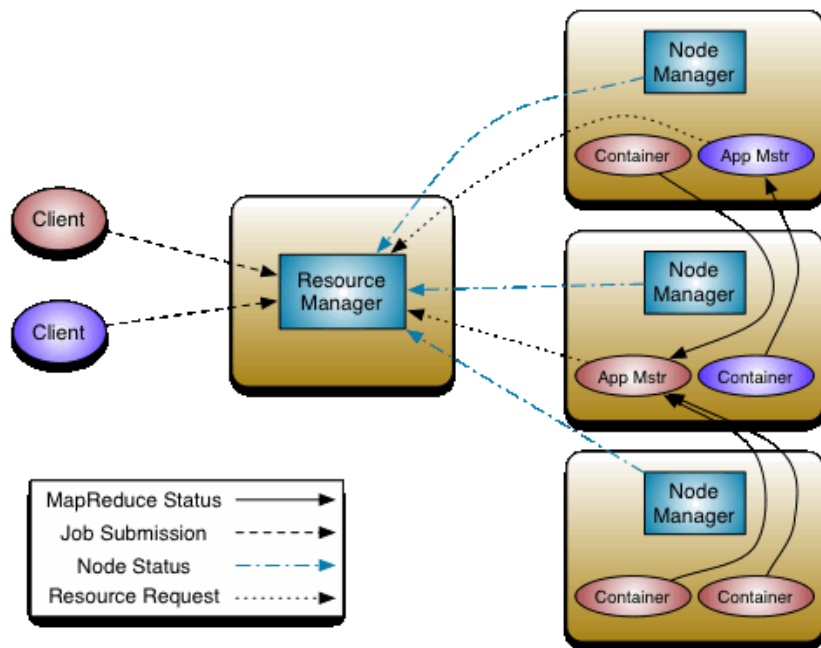
1) Client -> Resource Manager

   Submit App Master

2) Resource Manager -> Node Manager

   Start App Master

3) Application Master -> Resource Manager

   Request containers

4) Resource Manager -> Node Manager

   Start tasks in containers

5) Node Manager -> Resource Manager

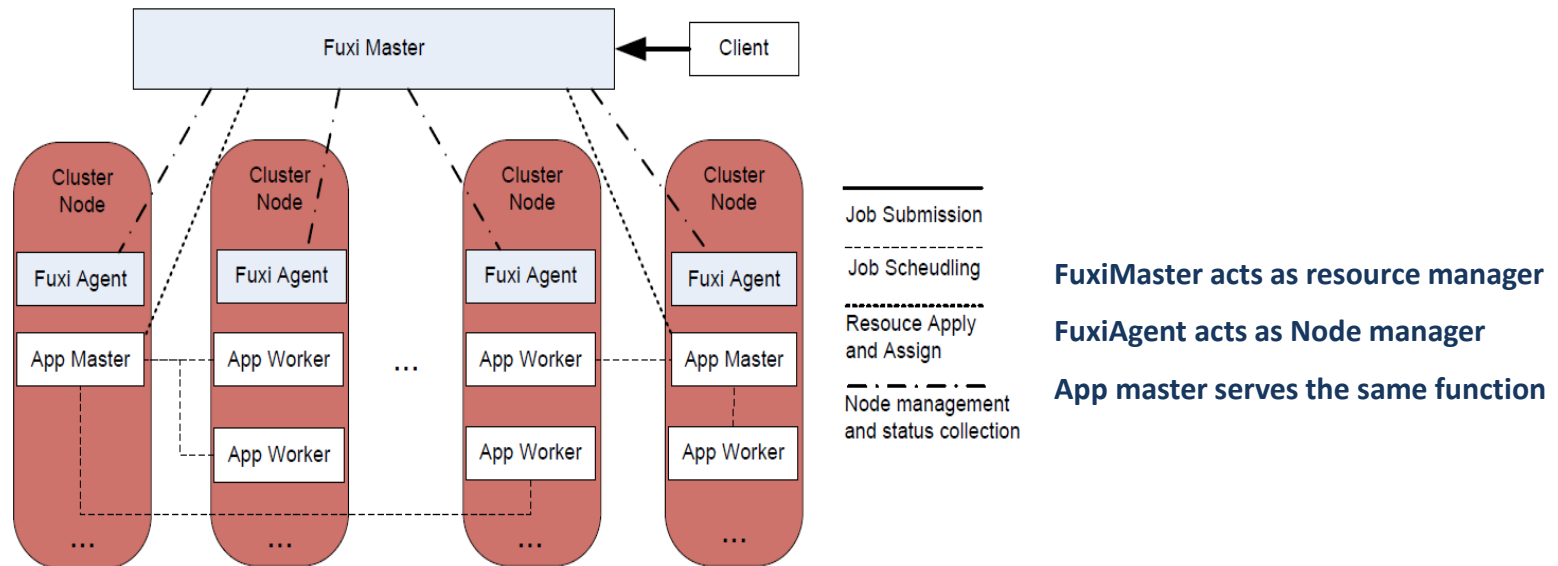 report running and *terminated container, trigger new round of scheduling.*

Limitations

- **Scheduling dimensions**
  - (only CPU and memory) are limited and not easy to extend.
- **Scalability issues**
  - Resource assignment at the granularity of a task instance
  - The allocated resource to each container has to be reclaimed by NM once it terminates even if the application has more ready tasks to run.
  - RM has to conduct additional rounds of rescheduling.
  - At most 4k-nodes.
- **Failover mechanism** is not extremely pool to support larger scale:
  - RM**:** Non-transparent Resource Manager Failover
    - merely recover and restore its own states. **AMs can not survive RM restart**.
  - NM&AM:
    - It **uses mandatory termination and simply re-dos** failed/killed applications leading to substantial wastes and overheads.

17

Fuxi System

- The Fuxi architecture has similarities to YARN

FuxiMaster acts as resource manager

FuxiAgent acts as Node manager

App master serves the same function

- Focus on two challenging problems:
  - *Scalability (+ Efficiency):*
  - How to support 5k or more nodes but avoiding message floods?
  - How to achieve hundreds of thousands of requests per second?
  - *Fault-Tolerance (+ Efficiency):*
  - How to provide transparent failover mechanism?

Improved Scalability

- **Incremental scheduling and communication**
  - Resource *request* is only sent once until the application master *release* the resources.
    - Scheduling tree and multi-level waiting queue with priority identified.
    - New round of scheduling is triggered only when resources release.
  - An incremental request will be sent only when the resource demands are dynamically adjusted.
    - Reducing frequency of message passing
    - Improving the whole cluster utilization

- **Multi-dimension resource allocation**
  - CPU, mem, other virtual resources
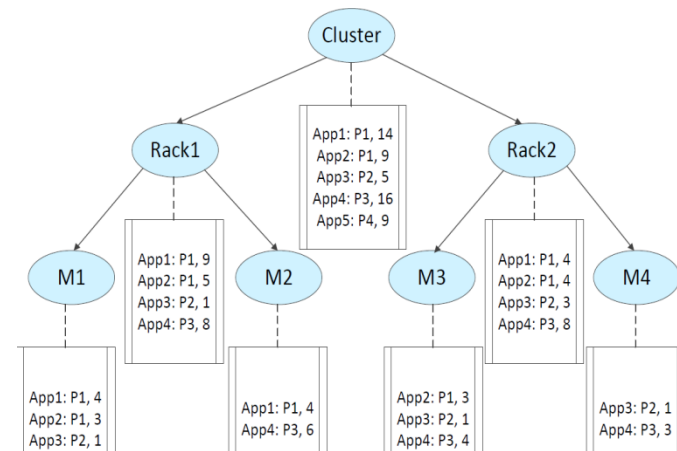
Scheduling tree example
(multi-level waiting queues)

19

Advanced Fault Tolerance

- **User-transparent failover mechanisms**
  - **Resource Manager:**
    - Refill *hard states* (meta-data, configuration file etc.) from light-weight checkpoint with no impact to running applications.
    - collect *soft states* from App Masters, Node Managers in run-time.
  - **Application Master:** can also performs failover to recover the finished and running workers by all task instances' snapshot.
  - **Node Manager:** rebuild the complete states with full granted resource and worker list.

- **Multi-level blacklist: "bad" (frequently failed) node detection**
  - Heartbeat threshold control
  - Application-level information collection
  - Plug-in service to aggregate OS-level information

Good News

- Collaborated Paper will appear in top international conference VLDB 2014

Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. *Fuxi: a Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale*, The 40th International Conference on Very Large Data Base (VLDB 2014), Hangzhou, China, 2014 (to appear)

Google's view of multiple framework mixture

monolithic scheduler

SCHEDULER
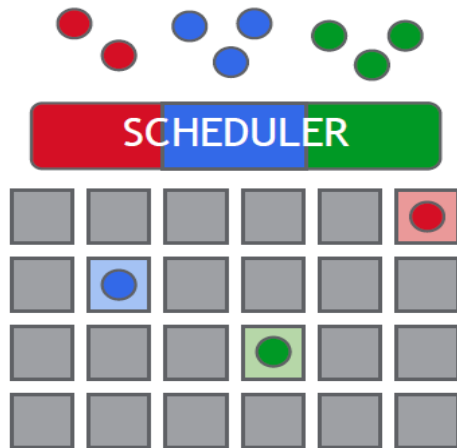
- hard to diversify
- code growth
- scalability bottleneck

static partitioning

S_0 S_1 S_2
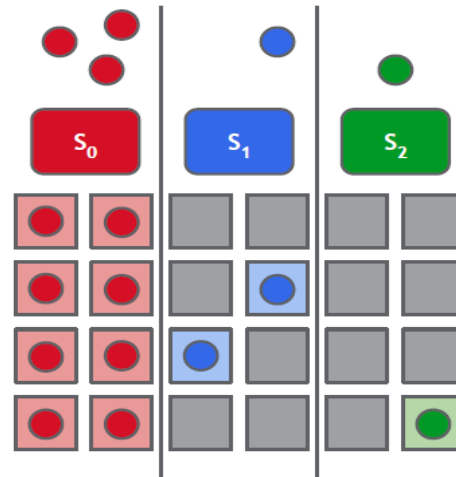
- poor utilization
- inflexible

two-level

S_0 S_1 S_2

RESOURCE MANAGER
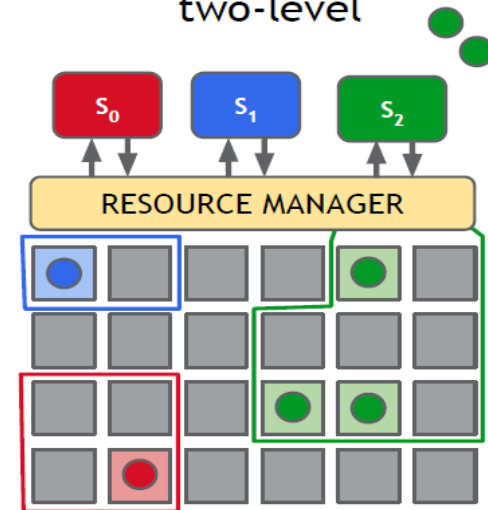
- hoarding
- information hiding

Mesos/Yarn/Fuxi

- Conservative resource-visibility and locking algorithms limit both flexibility and parallelism.
- Hard to place difficult-to-schedule "picky" jobs or to make decisions that require access to the state of the entire cluster.

Third Generation? – Google's view

- Google proposes ***shared state*** by using lock-free optimistic concurrency control.

  – Lock-free optimistic concurrency control over the shared states, collected timely from different nodes.

  – Each application framework holds the same resources view and competes for the same piece of resource with a central **coordination component** for arbitration.

Potential Limitations in Google's Omega

- More efficient? Let's wait and see…

- It is hard to enforce global properties such as capacity/fairness/deadlines.

- It may be tough for heterogeneous frameworks outside Google when sharing the same cluster.

- Current simulation-based evaluation needs more real-life experience and practice.

Scheduler Simulation

- Why we need Scheduler Load Simulation?
  - Evaluating in a real cluster is always time and cost consuming
  - it is also very hard to find a large-enough cluster.
  - It is quite useful predict how well a scheduler algorithm for some specific workload

- Examples:
  - GridSim
  - CloudSim
  - Yarn SLS
  - Etc.

Simulation Workflow

- UT(Unit Test) or ST(System Test)

Resource Manager

Scheduler Wrapper

real Scheduler

heartbeats

Simulator

Job

Node

handling and
dispatching NM/AMs he
artbeat events

All NM/AM simulators
run in a thread pool
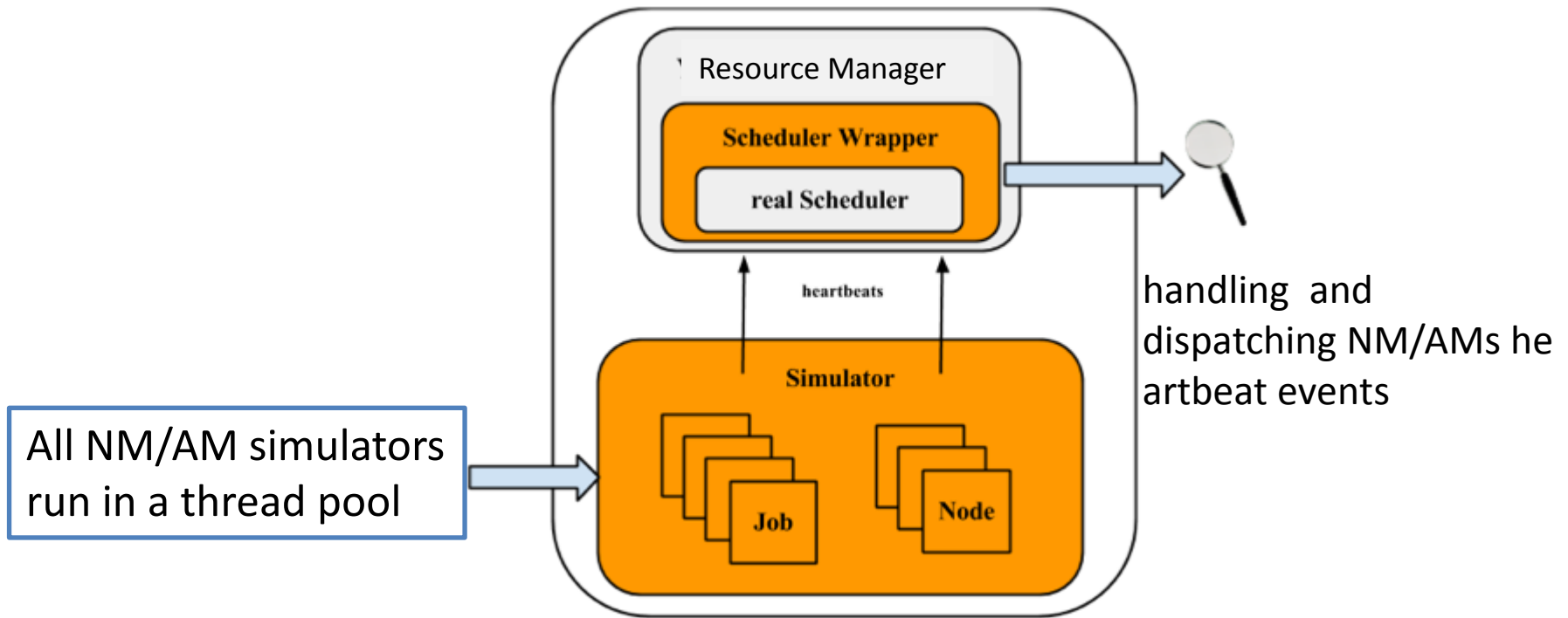
- Typical Resource Scheduler Models:
    - Node Manager is proactive
    - App Master is proactive

Methodology：Trace-driven Simulator Design

- Data Provenances:
  - Light-weighted synthetic workloads:
    - Sampled data;
    - Using parameters drawn from empirical workload distribution;
    - *Similar to what we do (Google trace, work from Beihang &Leeds)*
  - Real trace data:
    - Historical workload traces, e.g., job status tracelog
- Reflexed Metrics:
  - resource request size, tasks per job, job arrival/submission rate, task duration, related constraints

Methodology

- In small scale cluster, larger scale cluster environment could be simulated by creating Node Managers.

- Emulate app Masters and the interactions with Resource Manager
  - Resource request (when and how many)
  - Resource reclaim (when <- task duration)

- Work in progress and see more results in our paper to appear.

More Details

- Ask Renyu for more details of the design in distributed scheduler load simulator

- Email:
  - yangry@act.buaa.edu.cn
  - Renyu Yang @ Beihang

Reference

[1] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In Proc. SoCC, page 5. ACM, 2013.

[2] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes. Omega: Flexible, scalable schedulers for large compute clusters. In Proc. EuroSys, pages 351-364. ACM, 2013

[3] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In Proc. NSDI, Usenix, 2011.

[4] J. Dean and S. Ghemawat. Mapreduce: simplifed data processing on large clusters. Communications of the ACM, 51(1):107-113, 2008.

[5] Apache Hadoop. http://hadoop.apache.org/

[6] http://www.pdl.cmu.edu/SDI/2013/slides/wilkes-sdi.pdf

[7] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Jie Xu. Fuxi: Fault Tolerant Resource Management and Job Scheduling System at Internet Scale (to appear in Proceeding of VLDB 2014 vol.7) , Sept 2014.

[8] SLS: http://hadoop.apache.org/docs/r2.3.0/hadoop-sls/SchedulerLoadSimulator.html

Thank you for your attention!

Renyu Yang, Ph.D. student
School of Computing
Beihang University, Beijing, China
Homepage: http://act.buaa.edu.cn/yangrenyu
Email: yangry@act.buaa.edu.cn