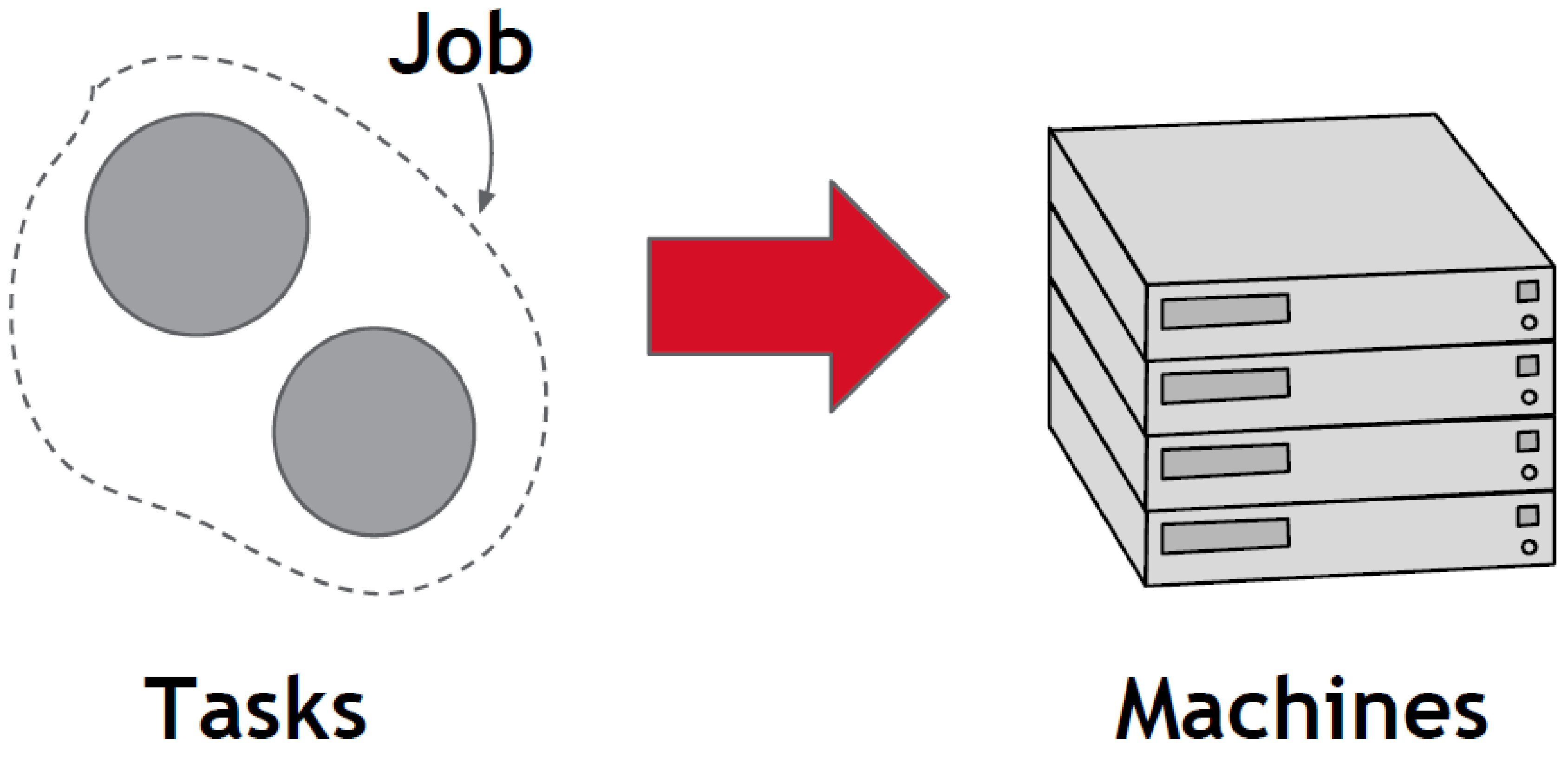
Architectural Evolution for Data Processing at Scale and Introduction to Alibaba’s ***Fuxi*** (伏羲) System

Renyu Yang and Jie Xu
COLAB (Beihang University and University of Leeds)
Supported by the National Basic Research Program of China (973)
under grant No. 2011CB302602

Resource Management and Job scheduling

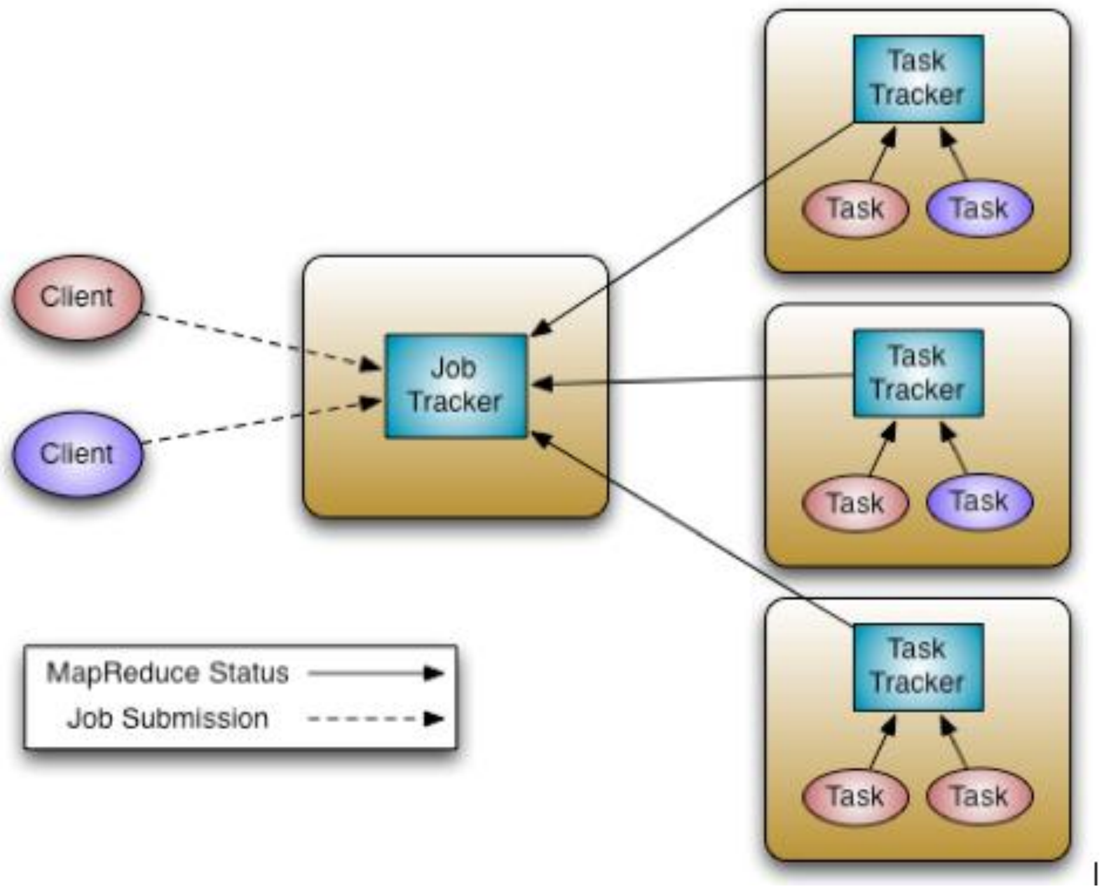


Hadoop MR v1 → Mesos → Yarn/Fuxi MR v2

Data Processing System Overview

- **1st Generation (G1)**:
  - Hadoop’s MapReduce programming paradigm
- **2nd Generation (G2)**:
  - Mesos
  - Yarn (MapReduce 2.0): “Yet Another Resource Negotiator”
  - Fuxi: Alibaba Cloud Inc’s Computing Platform
- **3rd Generation**？
  - Omega from Google

G1: Hadoop MR classic



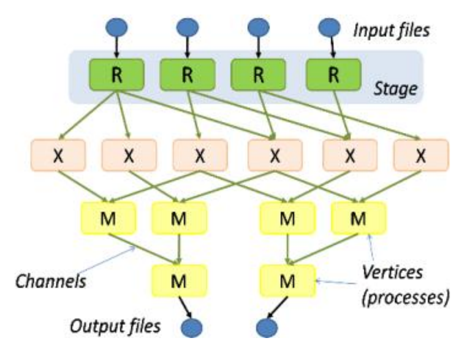
- JobTracker
  - Managing cluster resources
  - task scheduling
    - how to divide tasks
    - which node, how much resource for execution
- TaskTracker
  - Per-machine agent and daemon
  - Managing the execution of each individual task

Limitations

- Problems with large-scale resource management
  - > 4000 nodes
  - > 40k concurrent tasks
- Problems with resource utilization
- Overloaded JobTracker, and **single point of failure!**
- Support only one type of computing paradigm/framework (Slots only for Map or Reduce)
- Error recovery is very tricky due to complex state → weak failover mechanism

Increasing Requirement

- Rapid innovation in cluster computing frameworks

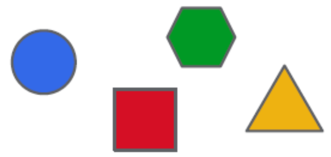


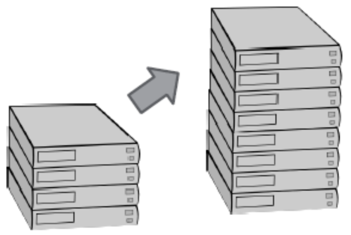
Dryad

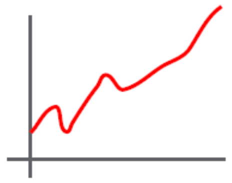
Pregel

Percolator

Trend observed !


Diverse workloads


Increasing cluster sizes

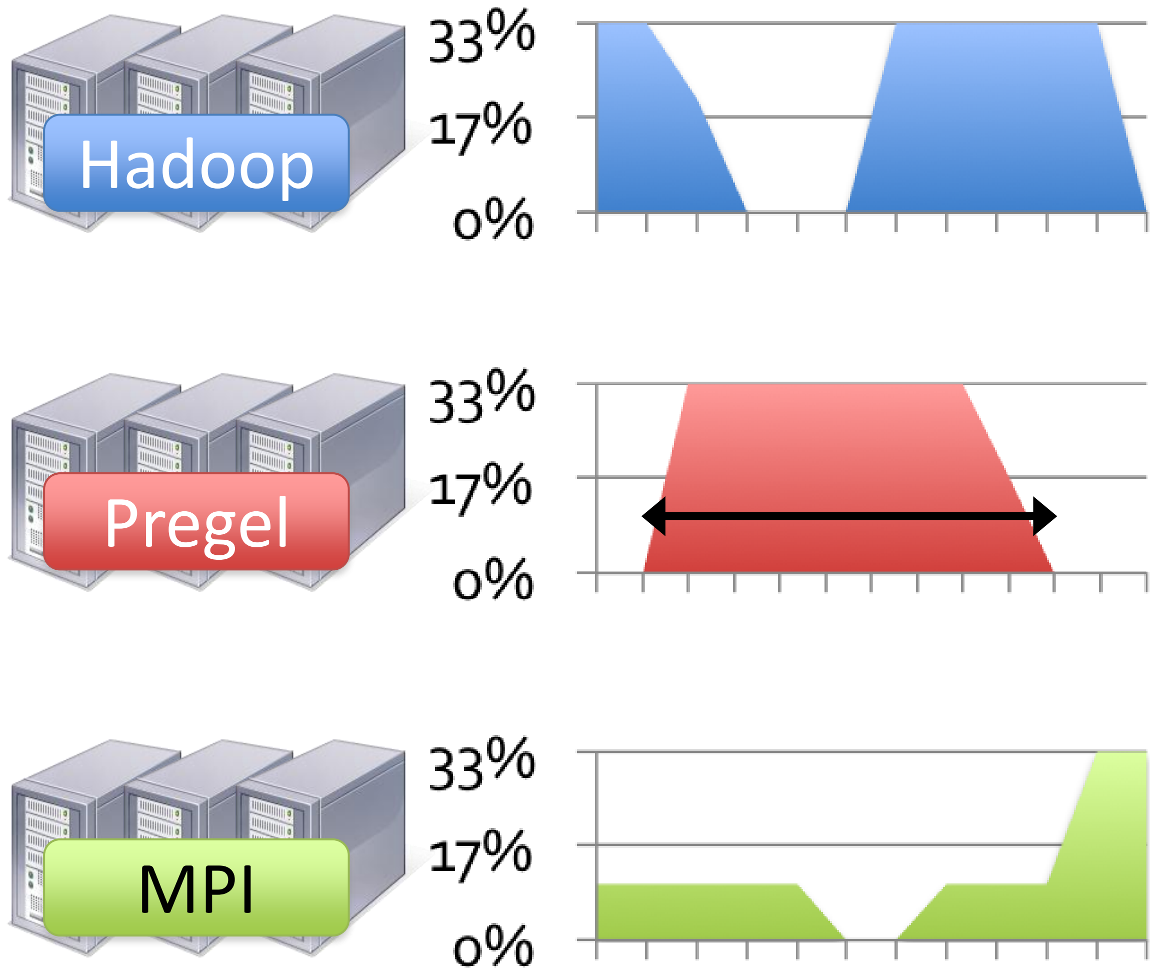

Growing job arrival rates

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

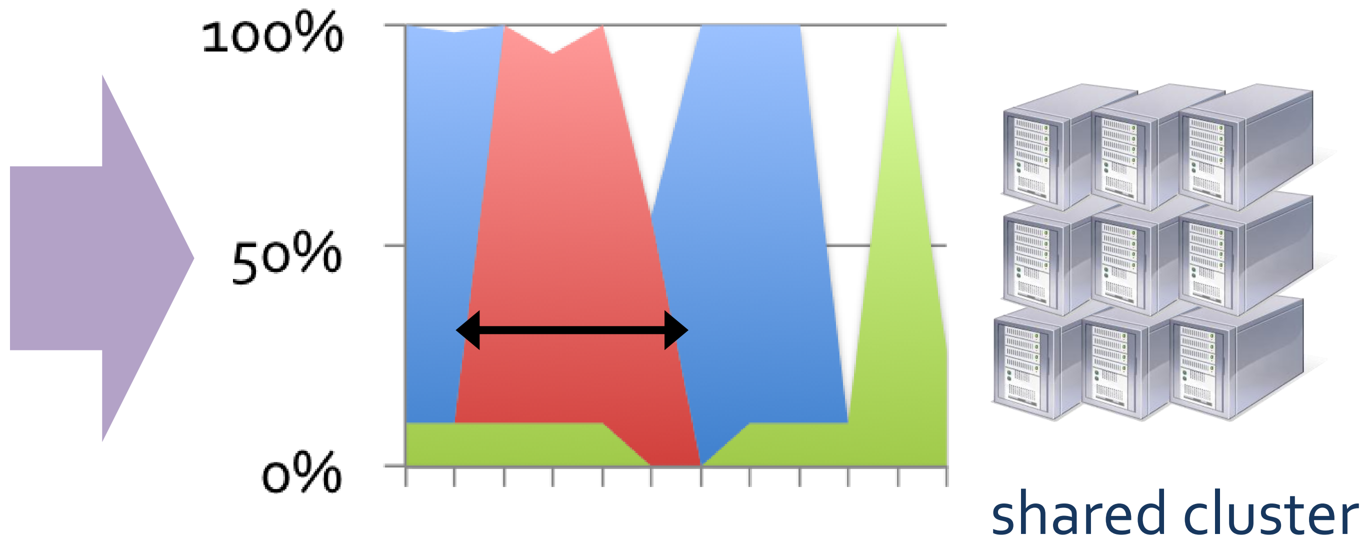
Motivation

No single framework optimal for resource utilization

Today: static partitioning



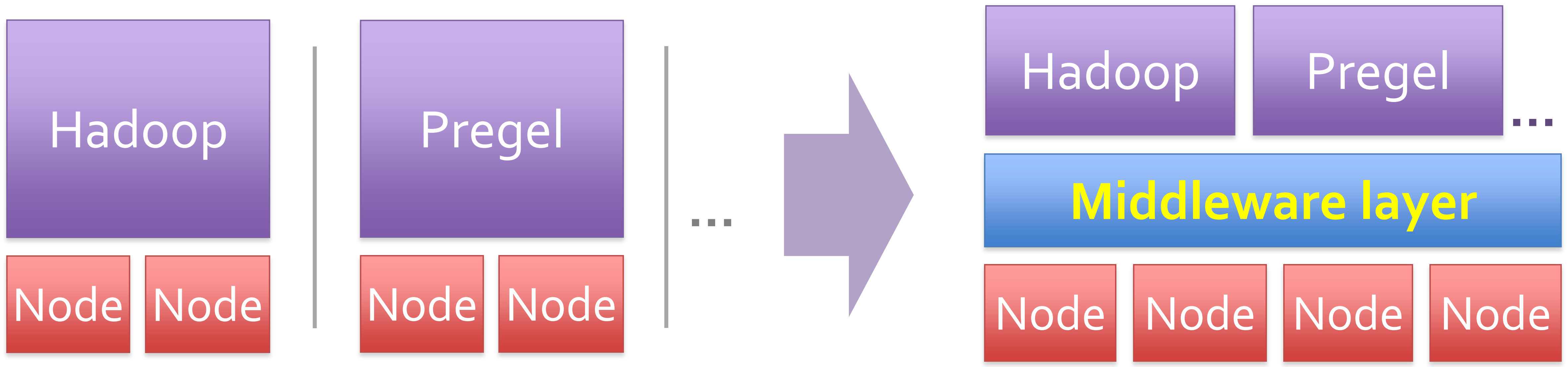
dynamic sharing



We would like to run multiple frameworks on a single but shared cluster
…in order to *maximize resource utilization*
…and if possible, to *share data* between frameworks

2nd Generation(G2) of Solutions

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

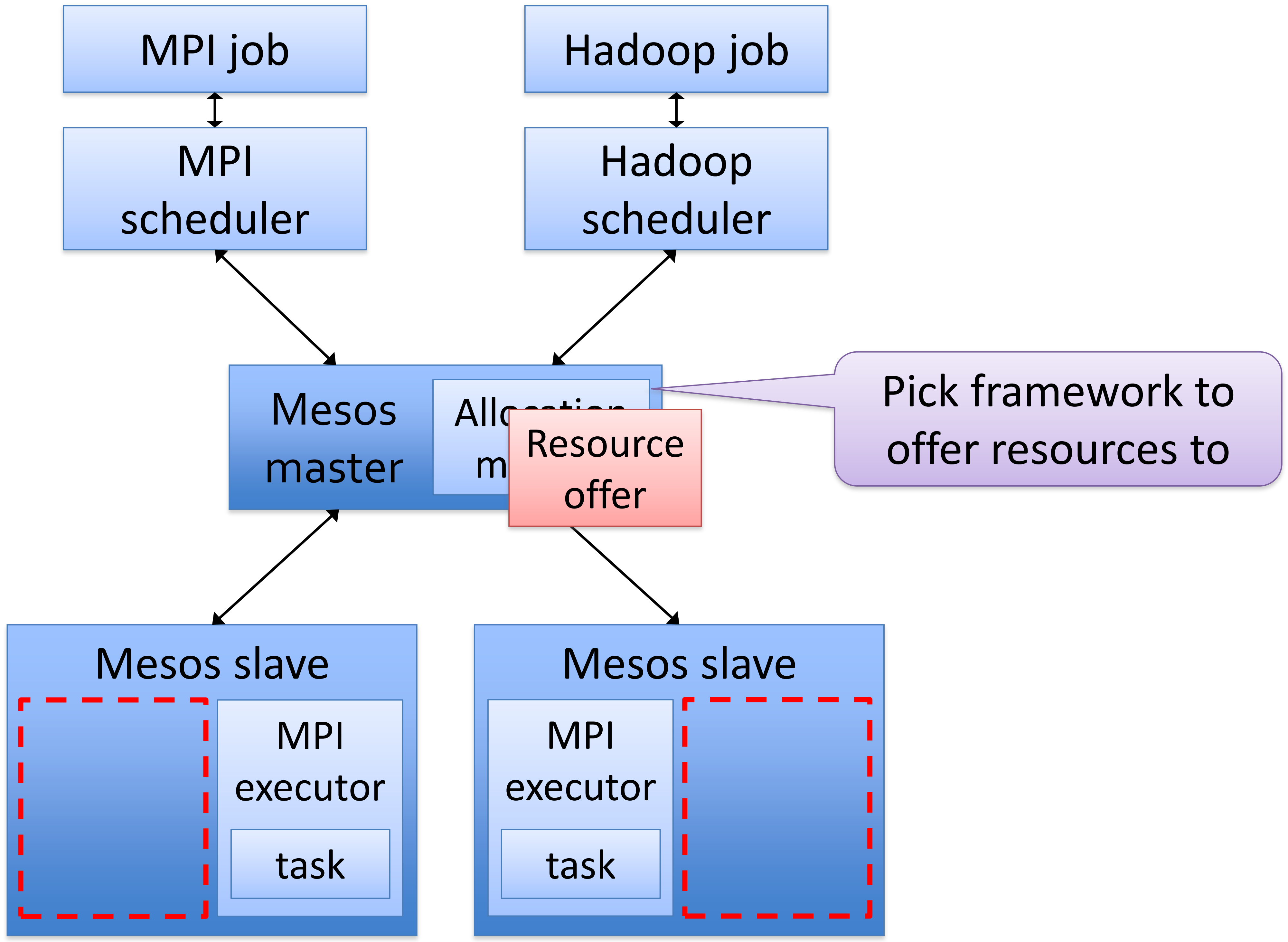


- **Goals**
  - **Higher utilization of resources**
  - **Support diverse frameworks**
  - **Better scalability up to 10,000+ of nodes**
  - **Improved reliability in face of failures**

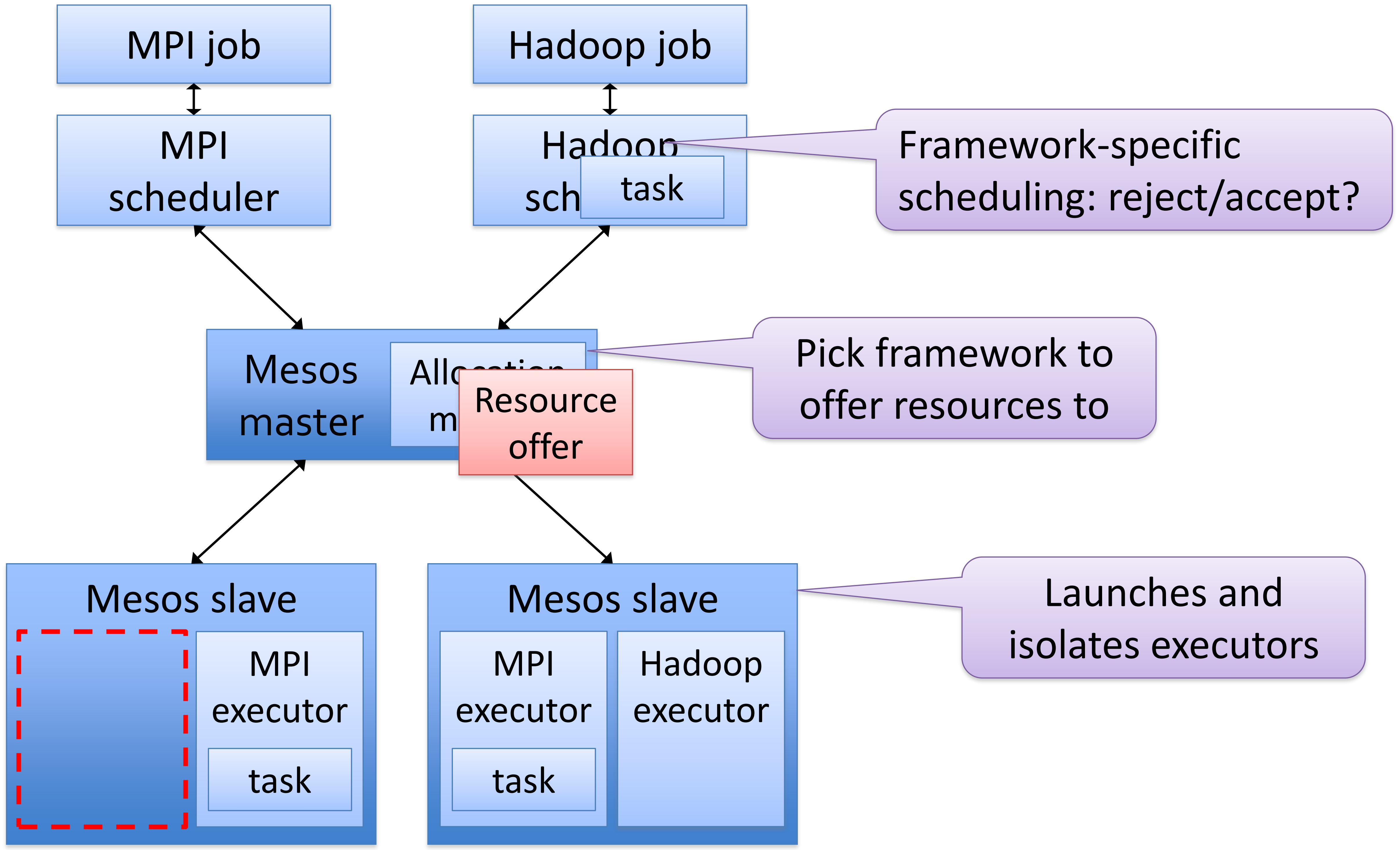
Mesos

- Mesos is designed based on a notion of "offer and accept" for resource management – the Mesos' **resource allocation module** decides how many resources should be offered to each application framework, based on an organizational policy such as fair sharing, while frameworks decide which resources to accept and which tasks to run on them.

Mesos Architecture



Mesos Architecture



Limitations

- Passive offer-based policy
  - cannot ask for resources based on various criteria including locations
  - 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

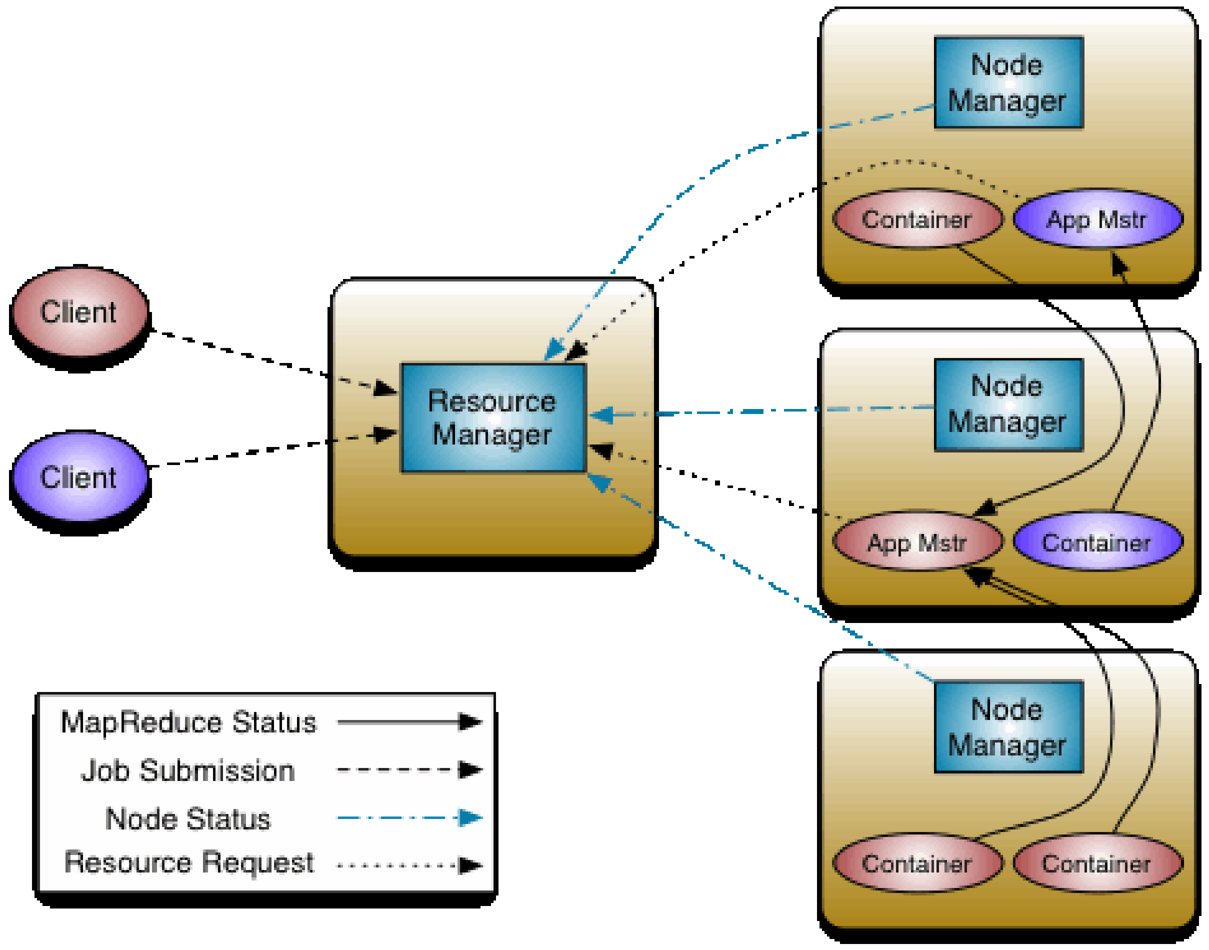
G2++: **Next Generation MRv2**

- De-coupling *JobTracker* into：
  - Resource Manager
  - Scheduling / Monitoring
- Following a request-based and active approach that
  - Improves scalability
  - Improves resource uitilization
  - Improves fault tolerance
- Providing slots for jobs and Map / Reduce

YARN

- De-coupling *JobTracker* 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
  - 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 and release containers

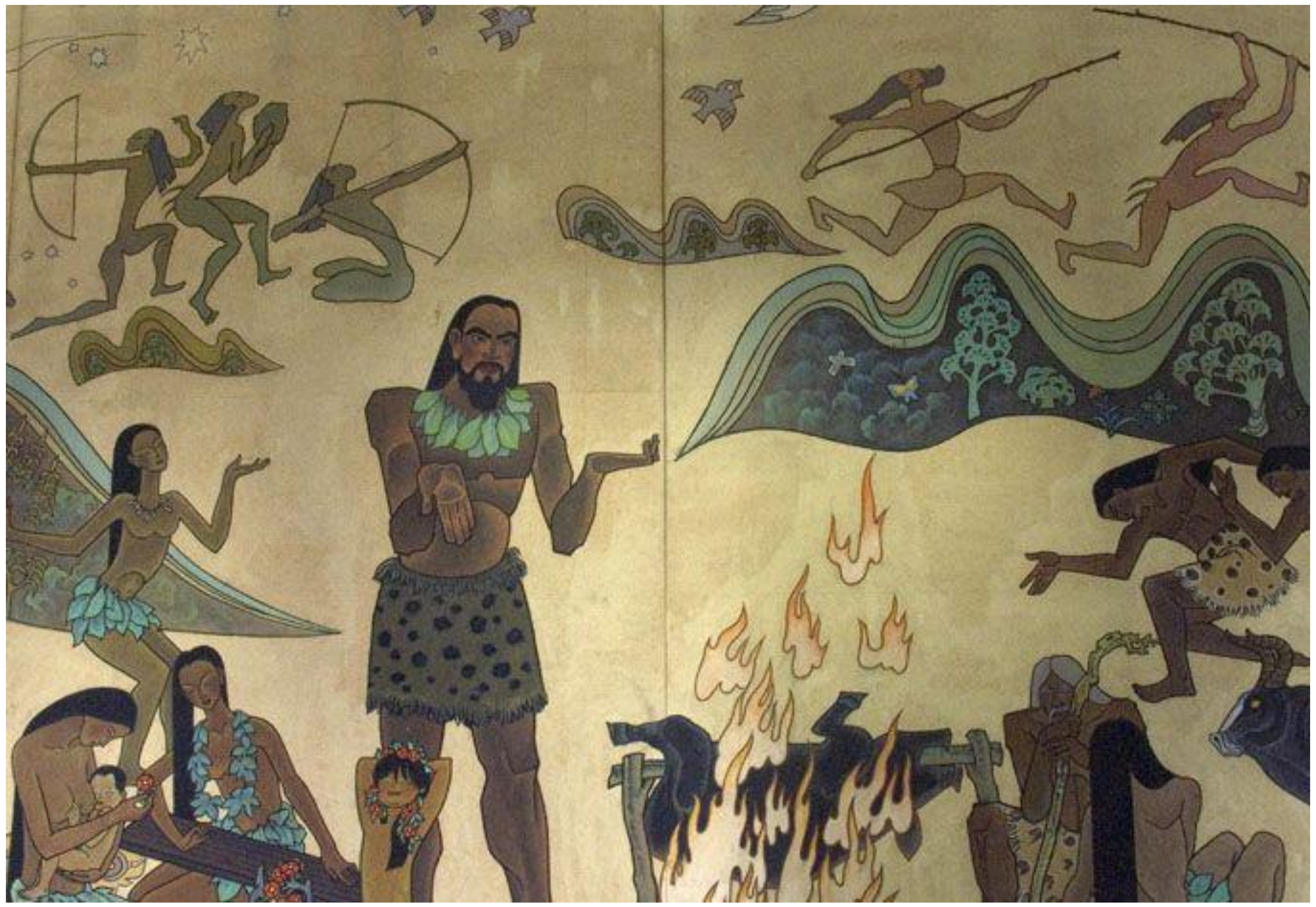
4) Resource Manager -> Node Manager

Start tasks in containers

Limitations

- **Scheduling dimensions** (only CPU and memory) are limited and not easy to extend.
- **Scalability issues**
  - Resource assignment at the granularity of a task (e.g. a Map task) and the allocated resource has to be returned from the application master, resulting in frequent requests and communications, thereby aggravating message floods.
  - At most 4k-nodes. *5k? 10k? Or more…*
- **Failover mechanism** for Resource Manager and Application Master is not efficient enough to support larger scale.
  - It uses mandatory termination and simply re-dos failed/killed applications.
  - Leading to substantial wastes and overheads.

Fuxi (伏羲)

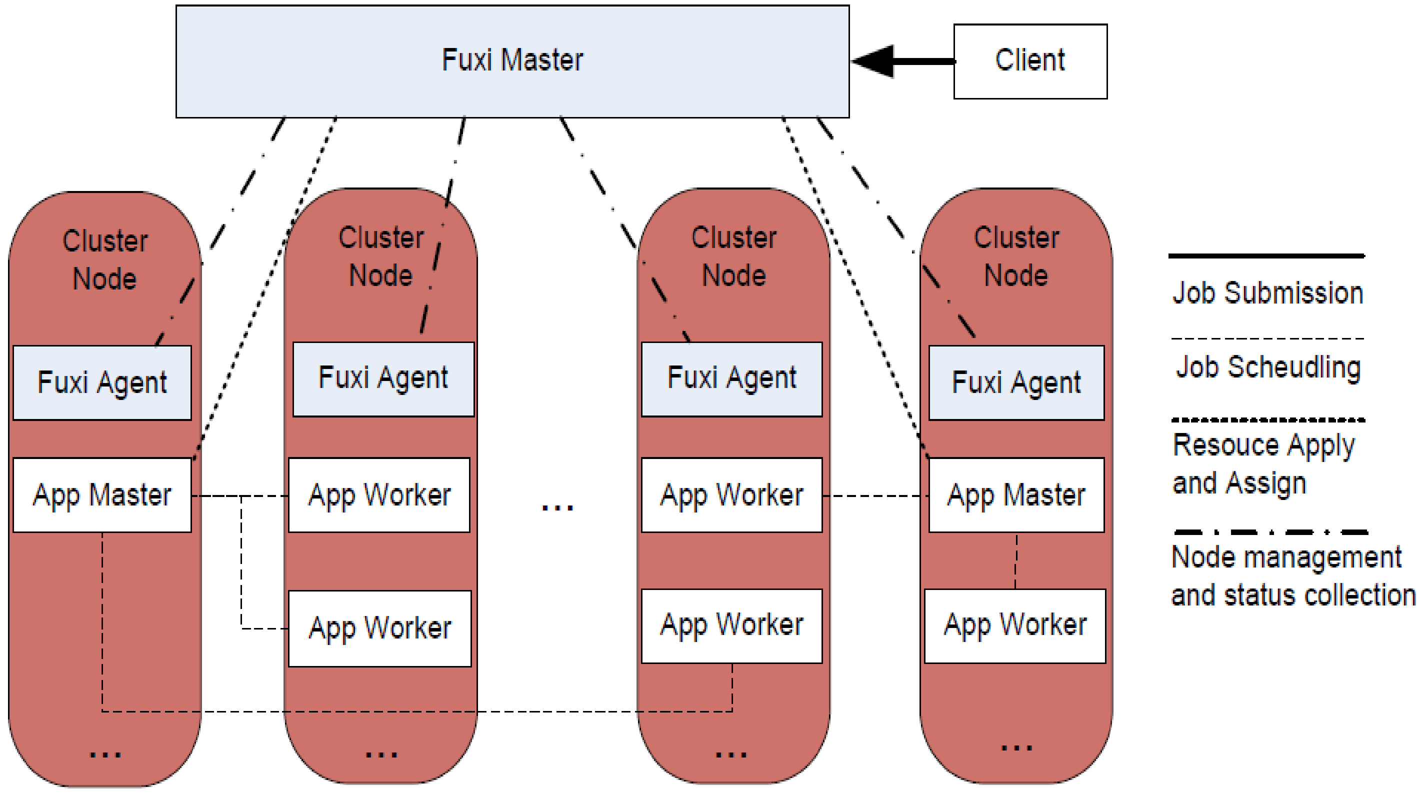


- 伏羲是我国古籍中记载的最早的王。伏羲聪慧过人，他根据天地万物的变化，发明创造了八卦。八卦可以推演出许多事物的变化，预卜事物的发展。

The name of our system, Fuxi, was derived from the first of the Three Sovereigns of ancient China and the inventor of the square and compass, trigram, Tai-Chi principles and the calendar, thereby being metaphorized into a powerful dominator in our system.

Fuxi System

- The Fuxi architecture has similarities to YARN



FuxiMaster acts as resource manager

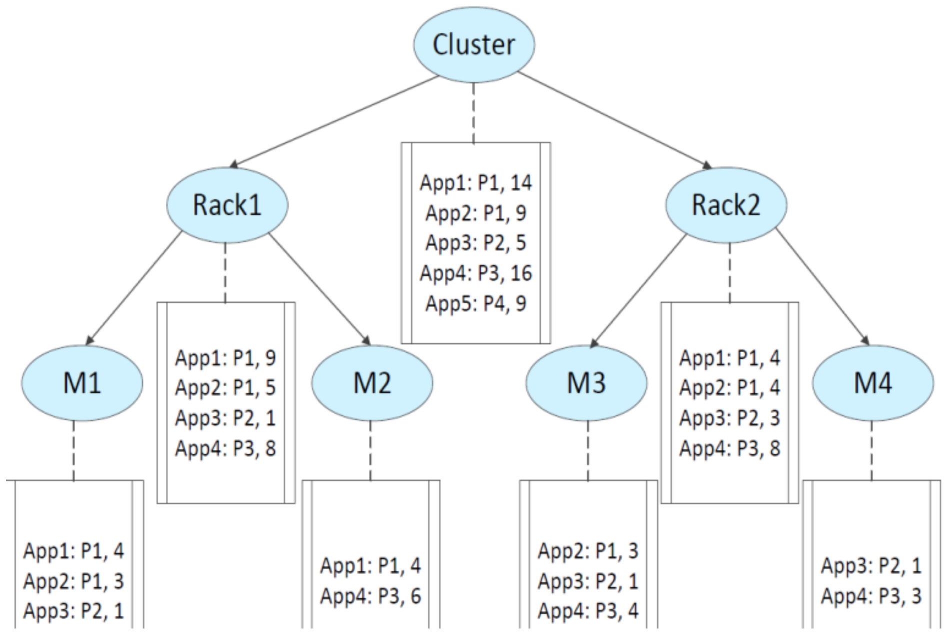
FuxiAgent acts as Node manager

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 resource request and allocation protocol**
  - Resource request is only sent once until the application master releases the resources.
    - Scheduling tree and multi-level waiting queue with priority identified
  - 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



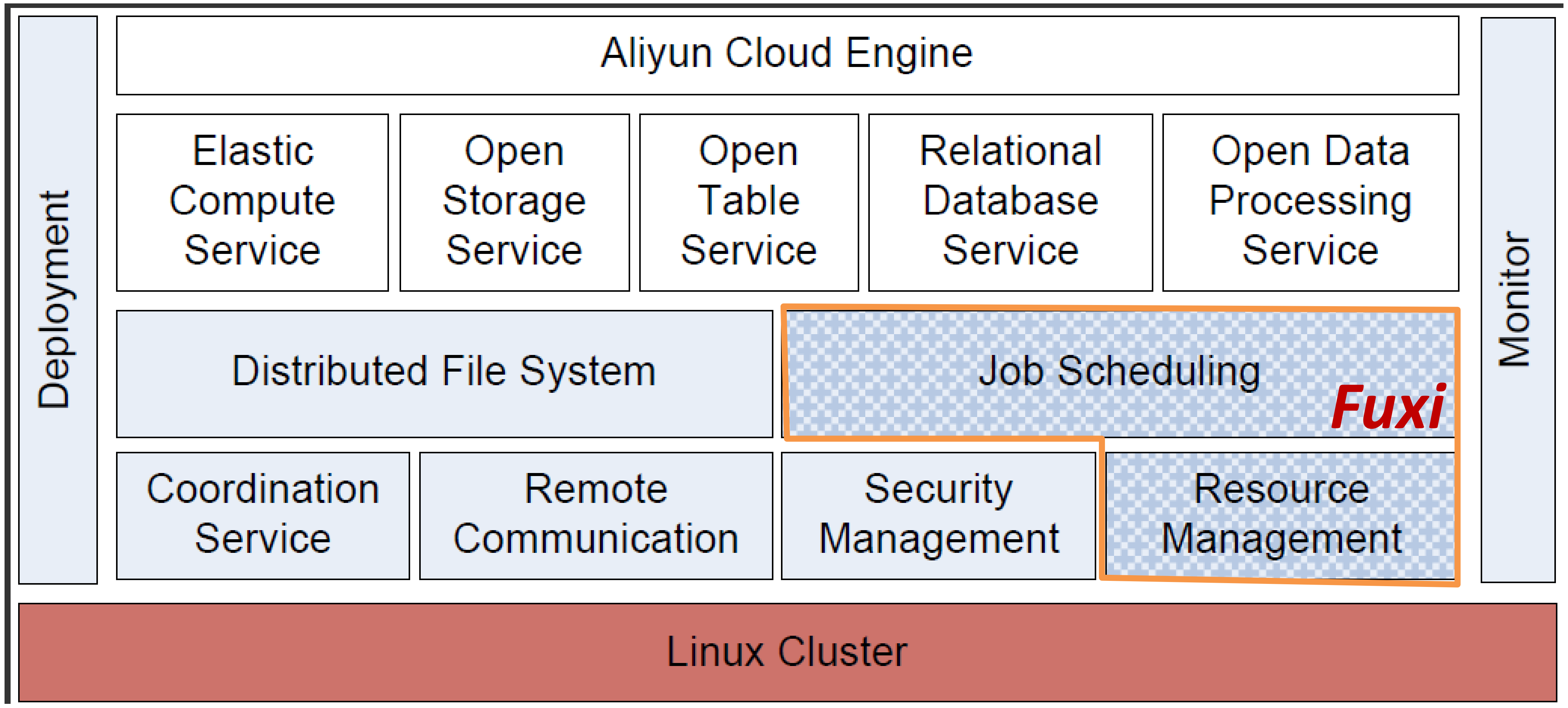
Scheduling tree example
(multi-level waiting queues)

Advanced Fault Tolerance

- **User-transparent failover mechanisms**
  - **Resource Manager:** uses a real-time resource assignment rebuild approach by meta-data light-weight checkpointing with no impact to running applications.
  - **Application Master:** can also performs failover to recover the finished and running workers by checkpointing.
  - **Node Manager:** existing processes will be taken over rather than killed after failover.

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

Alibaba’s Apsara Ecosystem



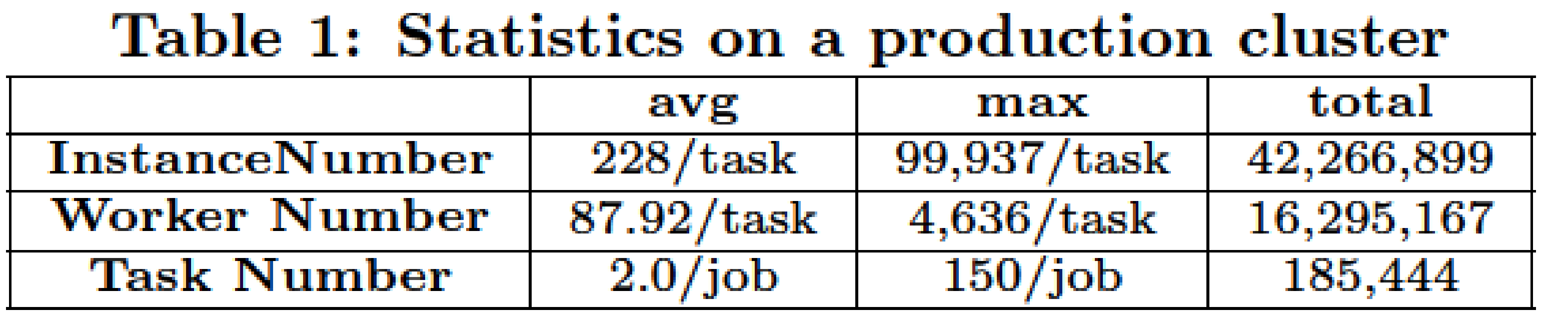


Table 1: Statistics on a production cluster

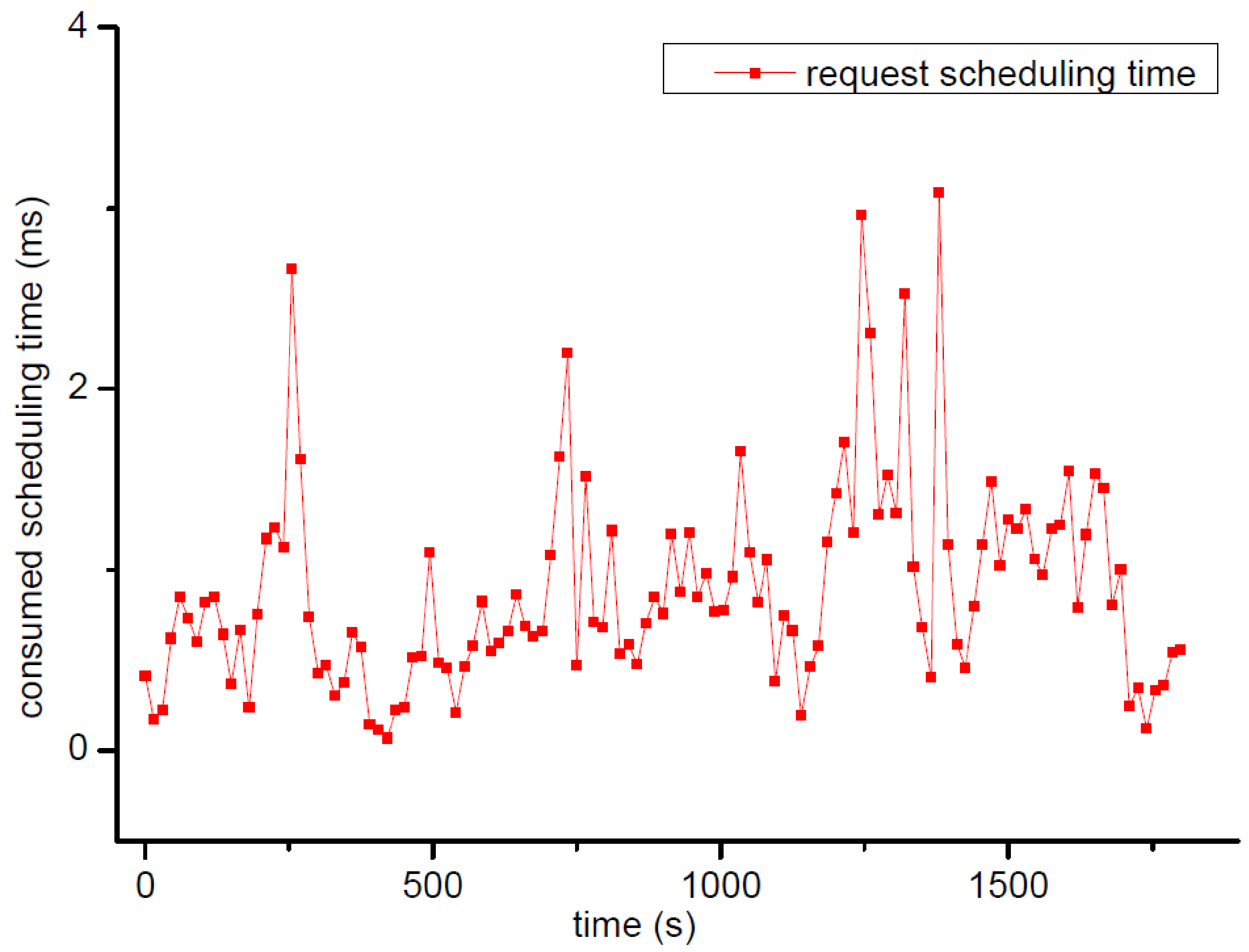
	avg	max	total
InstanceNumber	228/task	99,937/task	42,266,899
Worker Number	87.92/task	4,636/task	16,295,167
Task Number	2.0/job	150/job	185,444

Experimental Evaluation

- **Environment Setup**
  - **5,000 servers** with each machine consisting of 2.20GHz 6 cores Xeon E5-2430, 96GB memory and 12*2T disk.

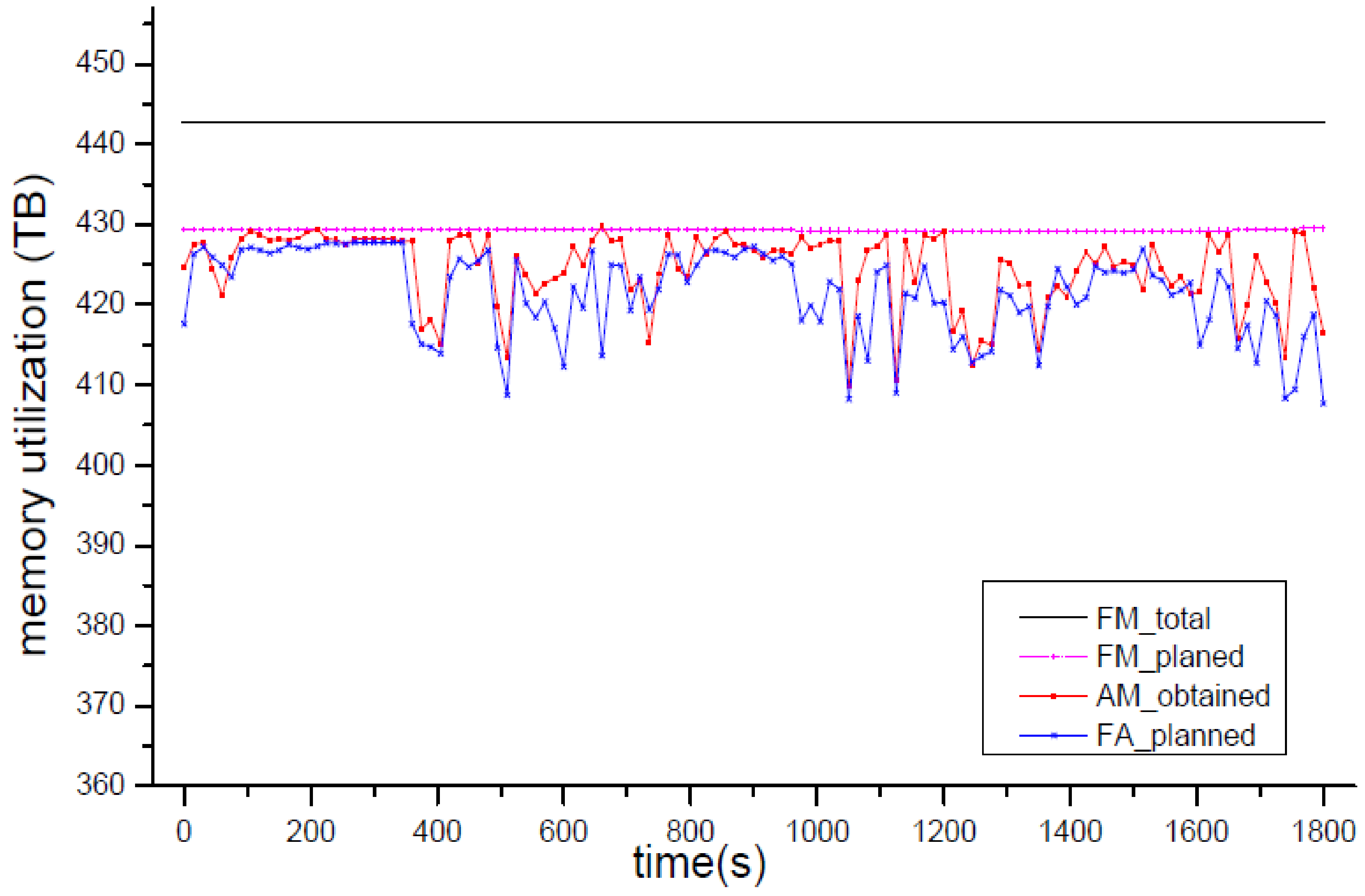
- **Objectives:**
  - **Scheduling performance** when 1,000 jobs are submitted simultaneously to a cluster.
  - **Sort benchmarks** are utilized to illustrate the performance and capability of task execution in Fuxi.
  - Several **fault-injection** experiments are carried out

Job Scheduling Time in Fuxi



The average scheduling time takes merely **0.88 ms** and the peak time consumption for scheduling is no more than **3 ms**, demonstrating that the system is rapid enough to reclaim the allocated resource as well as respond to incoming requests in a timely manner.

Resource Utilization in Fuxi



- Up to 97.1% of resources will be initially utilized by the scheduler.
- Gaps amongst these curves indicate overheads of processing requests by Fuxi master (In fact no more than 5%).
- The actual resource consumption by FuxiAgents is often less than requested due to user's overestimation.

Sort Benchmark

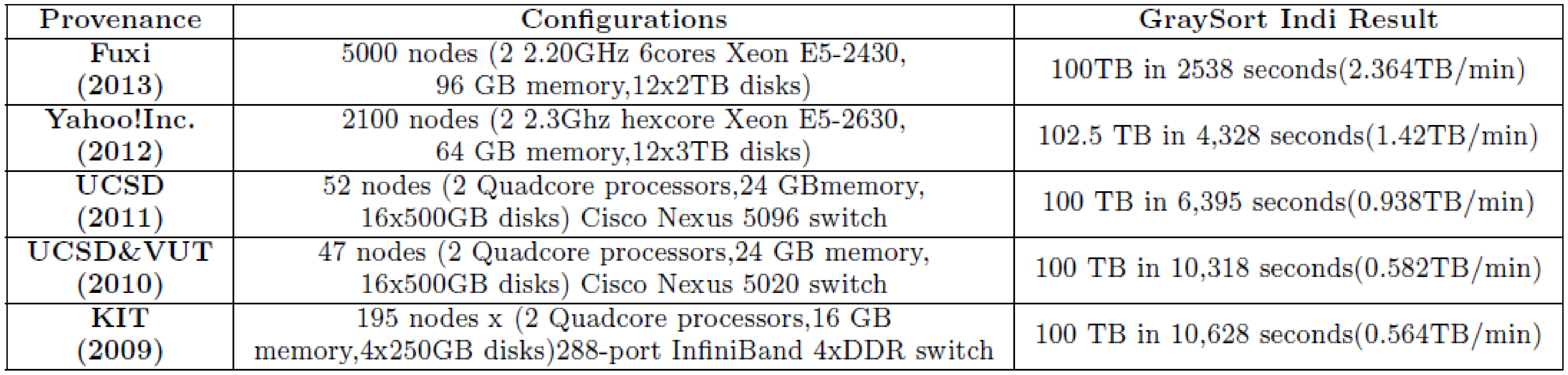


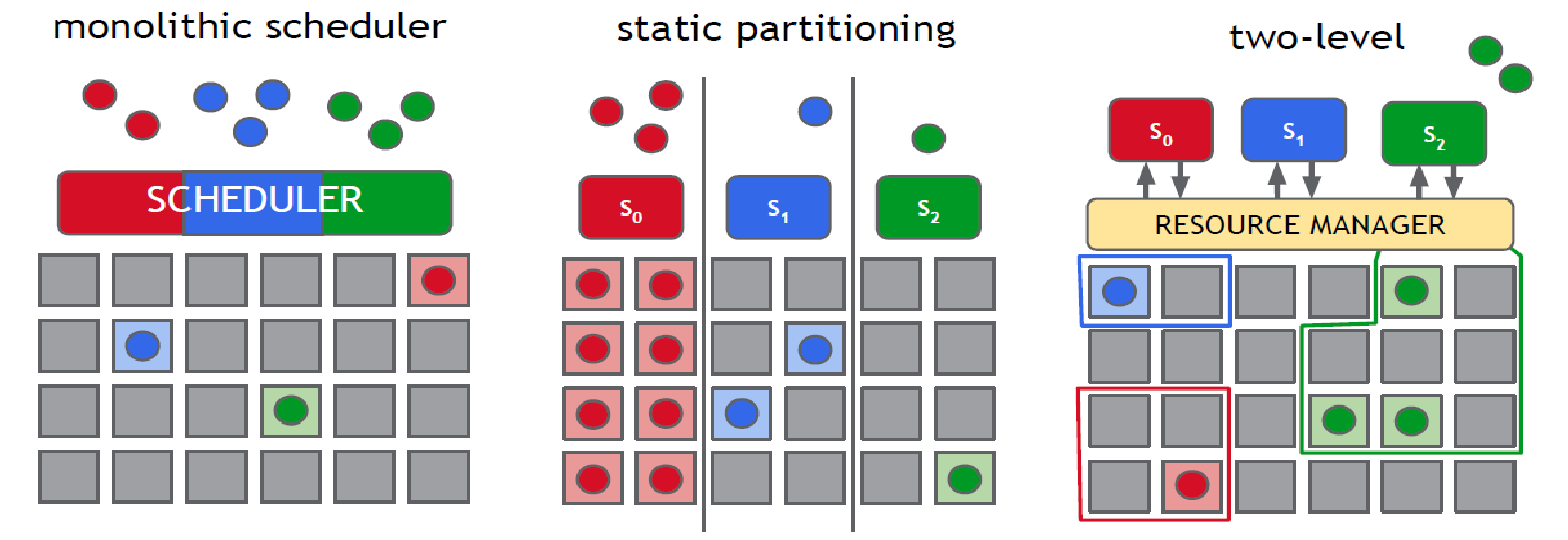
Table 2: GraySort Indi Result Comparison

Provenance	Configurations	GraySort Indi Result
Fuxi (2013)	5000 nodes (2 2.20GHz 6cores Xeon E5-2430, 96 GB memory,12x2TB disks)	100TB in 2538 seconds(2.364TB/min)
Yahoo!Inc. (2012)	2100 nodes (2 2.3Ghz hexcore Xeon E5-2630, 64 GB memory,12x3TB disks)	102.5 TB in 4,328 seconds(1.42TB/min)
UCSD (2011)	52 nodes (2 Quadcore processors,24 GBmemory, 16x500GB disks) Cisco Nexus 5096 switch	100 TB in 6,395 seconds(0.938TB/min)
UCSD&VUT (2010)	47 nodes (2 Quadcore processors,24 GB memory, 16x500GB disks) Cisco Nexus 5020 switch	100 TB in 10,318 seconds(0.582TB/min)
KIT (2009)	195 nodes x (2 Quadcore processors,16 GB memory,4x250GB disks)288-port InfiniBand 4xDDR switch	100 TB in 10,628 seconds(0.564TB/min)

End to end execution time in Fuxi is 2,538s which is equivalent to 2.364TB/minute, achieving 66.5% improvement, as compared with the most advanced Yahoo!'s Hadoop implementation

See more results (e.g. fault injection experiments) in our recent paper:
Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, and Jie Xu, "Fuxi: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale," (to appear in Proceeding of ***VLDB 2014 vol.7****) , Sept 2014.*

Google's view of multiple framework mixture



- hard to diversify
- code growth
- scalability bottleneck

- poor utilization
- inflexible

- 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.

References

[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: A Fault-Tolerant Resource Management and Job Scheduling System at Internet Scale **(to appear in VLDB 2014)**

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