Computing at Scale: Resource Scheduling Architectural Evolution and Introduction to Fuxi System

Renyu Yang（杨任宇）
Supervised by Prof. Jie Xu
Ph.D. student@ Beihang University
Research Intern @ Alibaba Cloud Inc.
Member of CoLAB(Collaboration of Leeds, Alibaba, Beihang)
2014.8.19 @Tsinghua University
Resource Scheduling Problems

Tasks → Job → Machines
Challenges: Computing at Scale

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

• Case studies:
  – System Architectural Evolution
  – Fuxi system
Resource Management System Overview (non-official)

- **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
Limitations

- Problems with large scale
  - > 4000 nodes
  - > 40k concurrent tasks
- Only support one type of computing paradigm (Slots only for Map or Reduce)
- Problems with resource utilization
- 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

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

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
2nd Generation(G2) Core Idea:

• **De-coupling the functionalities of JobTracker:**
  – Resource Management
  – Scheduling / Monitoring
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**
  - MPI scheduler
- **Hadoop job**
  - Hadoop scheduler

---

- **Mesos master**
- **Resource offer**
  - Pick framework to offer resources to

---

- **Mesos slave**
  - MPI executor
  - task

---

- **Mesos slave**
  - MPI executor
  - task

---

**Hadoop MR v1**

**Mesos**

**Yarn/Fuxi MR v2**
Mesos Architecture

MPI job
- MPI scheduler

Hadoop job
- Hadoop scheduler
- Task

Mesos master
- Resource offer
- Allocation

Mesos slave
- MPI executor
- Task

Mesos slave
- MPI executor
- Task

- Hadoop executor

Framework-specific scheduling: reject/accept?

Pick framework to offer resources to

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 -> Application Master
   response allocated containers

5) Application Master -> Node Manager
   Assign resources to tasks
   (assignment)
   Start tasks in containers
   (start Container -> stop container)

6) 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\(^1\).
Limitations

• **Failover mechanism** is extremely poor to support larger scale:
  
  – *RM*:
    • Non-transparent Resource Manager Failover
    • merely recover and restore its own states.
    • *AMs cannot survive RM restart*.
  
  – *NM&AM*:
    • It *uses mandatory termination*(mark running container to "killed")
    • **simply re-dos** failed/killed applications.
    • leading to substantial wastes and overheads.

• Possible reason: Yarn directly inherits from Hadoop1.0 in open-source community
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 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

Tree-based scheduling example (multi-level waiting queues)
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 from each app master.
Advanced Fault Tolerance

• “Bad” (frequently failed) node detection and multilevel blacklist.
  – Cluster-level
    • Heartbeat threshold control
    • Application-level information collection
    • Plug-in service to aggregate OS-level information
  – Task-level
    • If one task instance is reported failed on one machine, the machine will be added into the instance’s blacklist.
  – Job-level
    • When marked as “bad” by a certain number of instances, the machine will not be used by this job.

• Reduce the negative impact of faulty nodes on scheduling and decrease the probability of failures.
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
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.
• 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

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

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.
Publications

- Collaborated Paper will appear as the coming VLDB 2014 full paper

[1] 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

- 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.
Reference

Thank you for your attention!