# Proposal of Generic Scheduling Framework for YARN

Wei Shao@Rocket Fuel Inc

Last modified: June 19 2015

## Introduction

Currently, A typical YARN cluster has thousands of nodes, and runs many different kinds of applications: production applications, ad hoc user applications, long running services and so on. Different YARN scheduling policies may be suitable for different applications. For example, capacity scheduling can manage production applications well since application can get guaranteed resource share, fair scheduling can manage ad hoc user applications well since it can enforce fairness among users. However, current YARN scheduling framework doesn't have a mechanism for multiple scheduling policies work hierarchically in one cluster. So a generic scheduling framework is proposed here to address this limitation. The proposal tries to solve many other issues in current YARN scheduling framework as well.

## Issues of Current YARN Scheduling Framework

YARN-3306 talked about many issues of today's YARN scheduling framework, and proposed a per-queue policy driven framework. In detail, it supported different scheduling policies for leaf queues. However, support of different scheduling policies for upper level queues is not seriously considered yet. This proposal supports different policies for any queue consistently, and also solves other issues below:

1. Preemption related issues. YARN-3405 and YARN-3414 solved some related issues in fair scheduler. However, there still are many issues not solved yet. One of them is that preemption works for leaf queues only. For example, minimal share of parent queue is not respected, applications in the same queue will not trigger preemption among each other. Capacity scheduler may have similar issues. This proposal provides a hierarchical preemption model to handle all queues and applications consistently by the concepts of single application queue and deltaAllocation.

2. Configurations related issues. Currently, configurations at application level is not supported. This feature can be useful in some cases. For example: For satisfying SLA, the applications in capacity scheduling queue or guaranteed capacity scheduling (proposed in YARN-3807) queue can specify their own capacity. Examples of some other issues are, sum of queue's minimalShare can be above cluster or parent queue which will cause odd behaviors, the leaf queue has limits configurations for users, applications which introduces additional complexity. This proposal provides a hierarchical configuration model for all queues, users and applications consistently by the concepts of single user/application queue.

3. Design related issues. Currently, scheduling framework not only runs scheduling algorithms, but also manages resources/nodes status and performs resource allocation/preemption actions for applications. In this proposal, the scheduling framework is disassembled to SchedulerManager, ResourceManager (in narrow sense) and ApplicationManager. SchedulerManager only runs scheduling algorithms. This is a better design in terms of responsibility decoupling.

4. Scalability related issues. In the execution model of current scheduling framework, procedures of 'handling external events from nodes/applications', 'execution of scheduling algorithms' and 'resource allocation/preemption for applications' are coupled. For example, on node heartbeat, all three procedures need to be executed sequentially one after another. And each procedure is internally coupled as well. For example, the resource allocation for the applications has to be in sequential order, because the priority order of applications is dynamically updated after each allocation. In this proposal, three procedures are both internally and mutually decoupled, and support scalable parallelism.

5. YARN-3807 and YARN-3808 also talked about some limitations of particular scheduling policies (fair and capacity), and how generic scheduling framework proposed here can help to solve them.

### Generic Scheduling Framework Design

The proposed scheduling model has three objects SchedulerManager, ResourceManager (in narrow sense) and ApplicationManager. All objects are implemented as state machines.

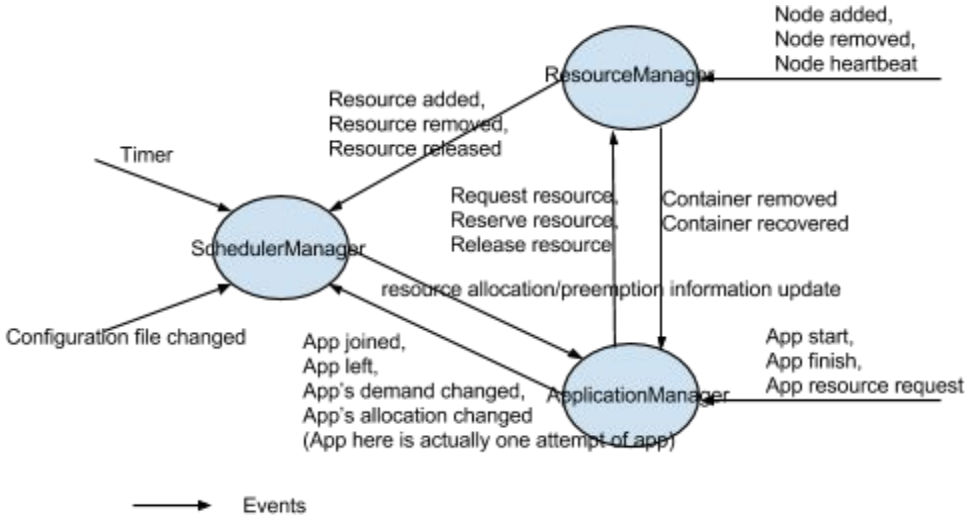


Figure 1

1. SchedulerManager: It runs scheduling algorithm periodically, the algorithm computes the amount of resources each application should preempt or can acquire based on current status of cluster and all applications. The algorithm can plug in different scheduling

policies. The algorithm results are sent to applications managed by ApplicationManager to make resources preemption/allocation decisions.

2. ResourceManager (in narrow sense): It manages resources/containers of cluster nodes. For example, it responses nodes heartbeat and update nodes information about allocated resources, available resources and reserved resources. It also isolates SchedulerManager/ApplicationManager from raw nodes events.

3. ApplicationManager. It manages applications, include ACLs, application attempts and so on. It also run an algorithm periodically for applications to perform resources preemption/allocation actions while respecting SchedulerManager inputs. The algorithm can plug in policies to avoid starvation and so on. It also isolates SchedulerManager from raw application events.

The states all three objects hold are soft states. It means, on failover, states can be restored by reports from cluster nodes and application masters. In this proposal, we will majorly talk about the design of SchedulerManager which is the most complicated one.

## SchedulerManager Design

The state machine design is introduced in terms of events, states and procedures triggered by events. The events are already shown in figure 1. The procedures of SchedulerManager are designed to be either update internal states or make external side effect, not both. And the procedures will call a generic scheduling policy interface which is implemented by specific policy like fair or capacity. So new scheduling policies only need to implement this generic interface. We will majorly talk about states, two types of procedures and the generic scheduling policy interface here.

**States**: The base scheduling unit is queue, the queues form a tree structure. Non leaf queue has an attached scheduling policy, leaf queue has an attached application. See figure 2 for an example. The queue's states are listed in table below.

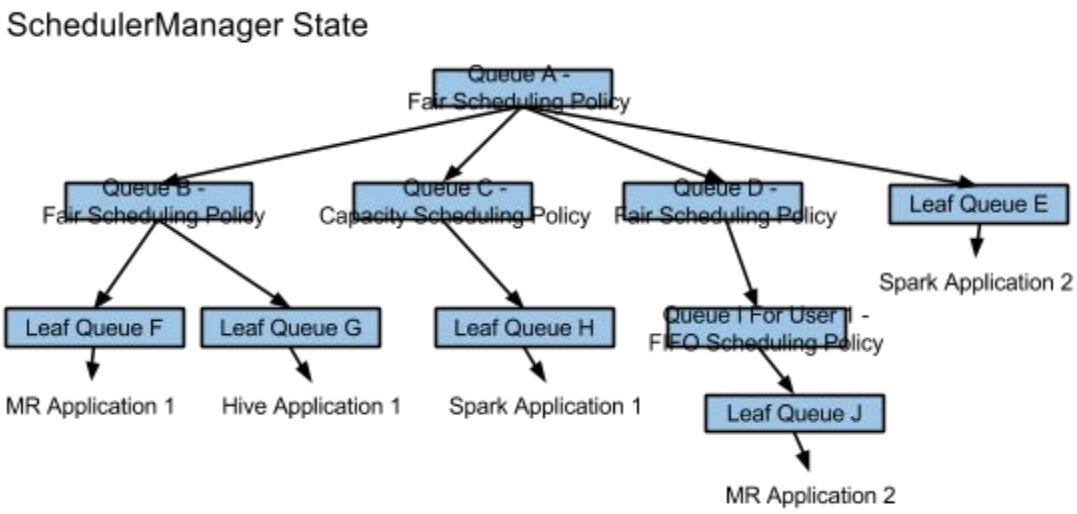


Figure 2

| State | Description |
|---|---|
| children | Sub queues managed by non-leaf queue. |
| pendingApplicationList | On application join event, new application will be added to the pendingApplicationList of the queue it specified. Only non-leaf queue has pendingApplicationList. |
| application | Leaf queue has an attached application, and has exact one. This is different from current YARN leaf queue which can manage multiple applications. Leaf queue is called ‘single application queue’ in this proposal, and will be created dynamically by the static queue specified in configuration file at runtime for each accepted application. The queue in configuration file can have configuration templates for its single application sub queues. Applications submitted to the static queue can specify a configuration template for its single application queue.<br>By introducing the concept ‘single application queue’, preemption model (support preemption among applications in same queue), resource allocation model, and configuration model (support application level configuration) which are introduced later can be implemented consistently for all queues and applications (since application is also a queue). |
| policy | Each non-leaf queue has its own scheduling policy to manage resources of children. For example, fair or capacity. |
| parameters | The input to the scheduling policy of the parent queue to make scheduling decisions for itself (one child of parent queue). There are four common parameters for all scheduling policies: |

| | |
|---|---|
| | configurations, demand, allocation and deltaAllocation. Particular scheduling policy can have its own parameters residing in children. |
| parameters.configurations | Configurations are the scheduling parameters which don't depend on running states of applications. For example, in figure 2, queue A uses fair policy, one configuration for fair scheduling is fairShare, so configurations of all sub queue B, C, D and E have item fairShare.<br>There are two kinds of configurations:<br>1. Static configurations: They are not changed at runtime. Usually, admin enforces cluster usage policies with static configurations, like fairWeight configuration of fair scheduling and capacity configuration of capacity scheduling.<br>2. Dynamic configurations: they can be updated at runtime, like fairShare configuration of fair scheduling. They have two states in updating process: expected and final (See updating procedure implementation for detail).<br>Particular scheduling policy can have its own configuration items.<br>Root queue doesn't have configurations.<br>Same configuration item can be either static or dynamic, like minimalShare in fair scheduling. It is dynamic in two cases:<br>1. MinimalThreshold instead of minimalShare is specified in config file. It is dynamic since its value is fairShare*minimalThreshold.<br>2. MinimalShare is specified as dynamic in config file with an initial value. It might be updated by scheduler to accommodate configurations of children. For example, set to the sum of minimalShare of children. |
| parameters.demand | Total resource demand of the queue. (resources can have locality preference.) |
| parameters.allocation | Current total resource allocation of the queue. |
| parameters.deltaAllocation | It is the amount of resources can be allocated to the queue if the value is positive, otherwise, it is the resources need to be preempted from the queue. (resources can have locality preference.) |

**Procedures Updating Internal States**: The procedures are listed in table below. All these procedures have following properties:

1. Traverse the tree with one of following patterns:
    a. Top down: updateStructure() and UpdateDeltaAllocations() visits queues in top down fashion.

b. Bottom down: updateDemands() and updateAllocations() visit queues in bottom up fashion.
c. Z shape: It has three steps, top down, bottom up and top down. updateConfigurations() works in Z shape fashion.

2. Update same states of the queue when each queue is visited in the traverse. For example, updateStructure() only updates children, pendingApplicationList, application and policy of each queue. updateDemands() only updates demand of each queue.

3. The traverse can be parallelized. For example, updateDemands() updates demand of all queues bottom up. In figure 2, demand updates of queues B, C, D and E can be executed in parallel before update of queue A. Parallelism is important for scheduling framework to scale up, since cluster could have thousands of running applications.

4. Idempotent. With this property, all the procedures can be triggered by timer event. For example, configurations should be updated when configuration file is changed, however, it can be triggered by timer event as well. In each timer event, procedures are executed in the order they listed in table below. Idempotence is also helpful for failover of resource management server.

5. Support transaction. In each timer event, procedures are executed in one transaction. For example, suppose in the execution, updateStructure() accepts a pending application, however, updateConfigurations() may reject it and cancel execution if sum of queues minimalShare is above cluster. When execution is cancelled, all the changes already made will be reverted. Admission control (whether application can be accepted or not) is implemented by this. Another example, updateConfigurations() may also cancel execution if invalid value is found in configuration file.

Idempotence and transaction support should be not hard to implement since all procedures just update internal states, don't make any external side effects.

| **Procedure Updating Internal States** | **Traverse Pattern** | **States to Update** | **Description** | **Required Functions in Generic Scheduling Policy Interface** |
|---|---|---|---|---|
| updateStructure() | Top down | children, pendingAppList, application, policy | The detail of implementation is in following paragraphs. | readConfigurations(), dispatchApplications(), |

| | | | | |
|---|---|---|---|---|
| updateParameters() | Call following 4 sub procedures in order. It can also send events to scheduling policy before and after each sub procedure. Particular scheduling policy can react as needed. | | | |
| updateDemands() | Bottom up | parameters. demand | The implementation ensures following invariants. For non leaf queue: $demand = \sum_{child} child.demand$ For leaf queue: demand = application.demand() | |
| updateAllocations() | Bottom up | parameters. allocation | The implementation ensures following invariants. For non leaf queue: $allocation = \sum_{child} child.allocation$ For leaf queue: allocation = application.allocation() | |
| updateConfigurations() | Z shape | parameters. configurations | The detail of implementation is in following paragraphs. | readConfigurations(), accommodateConfigurations(), adjustConfigurations() |
| updateDeltaAllocations() | Top down | parameters. deltaAllocation | The detail of implementation is in following paragraphs. | redistributeResources() |

The implementations of some procedures are:

1. updateStructure(): Each queue read configuration changes regarding queue structure and handle pending applications. The details are:

    a. Read its section in configuration file to update children (sub queues may be added/deleted in configuration file) and scheduling policy.

    b. Handling pending applications. One pending application may be chosen to run. New leaf queue will be created for the selected application. For example, in figure 2, queue B has hive application 1 in pendingApplicationList, if it is accepted, queue B will create leaf queue G for it. If the scheduling policy of the queue is configured to enforce one user per queue policy, it may add two queues for the selected application if it is the first accepted one submitted by the user. For example, in figure 2, queue D has MR application 2 in pendingApplicationList, if it

is accepted, queue D will create queue I for the user with FIFO scheduling policy, and leaf queue J for the application. (section for Queue D in configuration file specifies the default scheduling policy for users is FIFO.)

2. updateConfigurations(). The steps are:

    a. Top down initialization: Each queue updates children's configurations, and set all dynamic configuration items to expected state.

    b. Bottom up reporting: Each queue updates its own dynamic configuration items according to the configurations of children. For example, in figure 2, since both queue A and B use fair scheduling policy, suppose minimalShare of leaf queue F is 100G and minimalShare of leaf queue G is 200G, if minimalShare of queue B is dynamic with initial value 100G, it will be set to 300G according to formula below (state is still expected).

$$minimalShare = \max(minimalShare.initialValue, \sum_{child} child.minimalShare)$$

    c. Root queue adjustment: Root queue adjusts children's dynamic configuration items and set their states to final after adjustments. For example, in figure 2, suppose minimalShare of of queue B is 300G - dynamic, minimalShare of queue C is 500G - static, minimalShare of of queue D is 100G - static, minimalShare of of queue E is 100G - static, and the whole cluster has 900G memory (nodes may be added/removed). So root queue will decrease minimalShare of queue B to 200G, and also set its state to final.

    d. Top down adjustment: Each queue adjusts dynamic configuration items of children or force out some children based on both static configurations and adjusted final dynamic configurations of itself. It also sets all dynamic configuration items of children to final after adjustments. For example, in figure 2, suppose minimalShare of leaf queue F is 100G and minimalShare of leaf queue G is 200G, if final minimalShare of queue B is 200G, the queue has to force out one sub queue based on entering queue time or priority. Another example, in figure 2, suppose fairWeight of leaf queue F is 1 and fairWeight of leaf queue G is 2, if final fairShare of queue B is 300G, the queue will set fairShare of leaf queue F to 100G, and fairShare of leaf queue G to 200G.

3. UpdateDeltaAllocations(). The steps are:

    a. Root queue initialization: Root queue set deltaAllocation with formula

$$deltaAllocation = \min(resourceManager.freeResources, parameters.demand)$$

    resourceManager.freeResources has two kinds of resources:

        i. Available resources reported by heartbeats from nodes.

ii. Resources of the containers that are just killed (reported by ApplicationManager), but the heartbeats from corresponding nodes aren't received by ResourceManager yet, these resources will be available very shortly.

b. Top down update: When each queue is visited, it will:

i. Calculate preemption demands of all children. For example, in figure 2, the scheduling policy of queue A is fair scheduling, the preemption demand of each child is calculated based on formula below:

$$preemptionDemand = \max(0, \min(configurations.minimalShare, demand) - allocation)$$

ii. If

$$queue.deltaAllocation < \sum_{child} child.preemptionDemand$$

the queue will try to enforce invariants

$$queue.deltaAllocation = \sum_{child} child.deltaAllocation$$

and

$$\forall\, i \in children\ of\ positive\ preemptionDemand, \quad i.deltaAllocation = i.preemptionDemand$$

by distributing resources from the children who can offer resources immediately (for example, allocation > fairShare) to the children who need resources immediately (preemptionDemand > 0). In figure 2, suppose A.deltaAllocation = 5 G (cluster has some free resource), B.preemptionDemand = 10 G, C.preemptionDemand = 20 G. Possible updated deltaAllocations of children are: B.deltaAllocation = 10 G, C.deltaAllocation = 20 G, D.deltaAllocation = -10 G and E.deltaAllocation = -15 G.

iii. Else if

$$queue.deltaAllocation > \sum_{child} child.preemptionDemand$$

the queue will fulfill all preemption demands first, and then distribute left resources to satisfy regular demands. In figure 2, suppose A.deltaAllocation = 100 G (cluster has plenty of free resource), B.preemptionDemand = 10 G, C.preemptionDemand = 20 G. Possible updated deltaAllocations of children are: B.deltaAllocation = 15 G, C.deltaAllocation = 25 G, D.deltaAllocation = 35 G and E.deltaAllocation = 25G.

**Procedure Making External Side Effects**: There is only one such procedure. In each timer event, it is executed after all procedures updating internal states. It sends deltaAllocation to all running applications, and is idempotent as well. The implementation is:

```
updateDeltaAllocationOfApplications():
  for each leafQueue:
```

```
    leafQueue.application.setDeltaAllocation(
      leafQueue.parameters.deltaAllocation)
```

In summary, SchedulerManager executes scheduling algorithm in timer event. The workflow of scheduling algorithm is in figure 3.

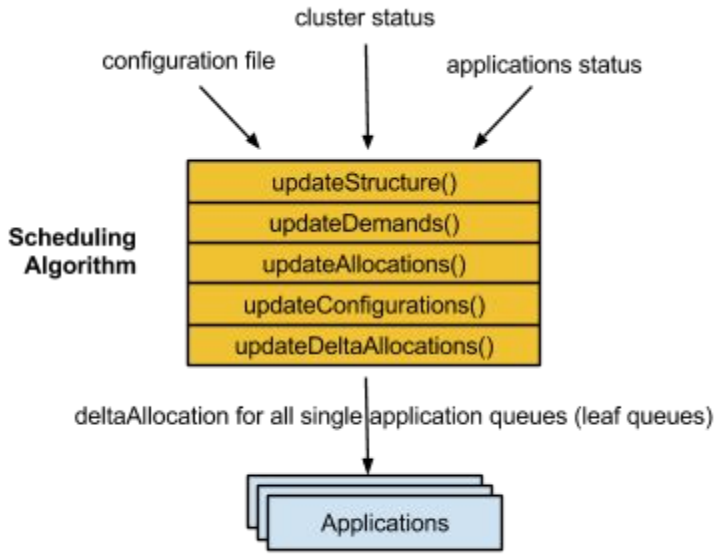


Figure 3

ApplicationManager will execute preemptResource() of all applications periodically, parallelism is supported. The implementation of preemptResource() is:

```
preemptResource()
  if application.deltaAllocation < 0:
    // Application may preempt more than one container.
    // Resources of containers in warning state should be
    // counted as resources preempted already.
    // The resources can be preempted might be less than
    // -deltaAllocation.
    preemptedResources = application.preemptResource()
    application.deltaAllocation += preemptedResources
```

ApplicationManager object will also execute allocateResource() of all applications periodically, parallelism is supported. Some policies can be adopted to avoid starvation (like round-robin). The implementation of allocateResource() is:

```
allocateResource():
  if application.deltaAllocation > 0:
    // Application can get/reserve resources from
    // ResourceManager object.
```

```
// Application will allocate resources for at most one
// container.
allocatedResources = application.allocateResource()
application.deltaAllocation -= allocatedResources
```

**Generic Scheduling Policy Interface**: The functions of generic scheduling policy interface are in table below. New scheduling policies should implement this interface.

| Functions | Description |
|---|---|
| readConfigurations() | Reads configuration items for children from external source like file. For example, in figure 2, queue A will read fairWeight of queue B, C ,D and E and update configurations.fairWeight of these children accordingly. |
| dispatchApplications() | Decides whether the queue can accept pending applications to run, select the application to run, and also constructs queue structure for the selected application. |
| accommodateConfigurations() | Checks whether the configurations of the queue is compatible with configurations of the children. And updates dynamic configuration items of the queue according to the configurations of children. For example, in figure 2, root queue A uses fair scheduling. Queue B use fair scheduling, the configurations of queue B and its children are not compatible if following invariant is broken. $configurations.minimalShare(static) \geq \sum_{child\ who\ has\ static\ minimalShare} child.configurations.minimalShar$ Queue C uses capacity scheduling, the configurations of queue C and its children are not compatible if following invariant is broken. $configurations.minimalShare(static) \geq \sum_{child} child.configurations.capacity$ if minimalShare of queue B is none (not set in config file) with state expected, it will be set to: $minimalShare = \sum_{child\ who\ has\ static\ minimalShare} child.minimalShare$ The queue and its parent may use different scheduling policies, so this function implemented by particular scheduling policy should supports all other types of scheduling policies. |
| adjustConfigurations() | Updates dynamic configuration items of children or may force some children out according to the configurations of the queue. For example, in figure 2, queue B will set FairShare of leaf |

| | |
|---|---|
| | queue F and G according to the following invariants.<br>$$\forall\, i, j \in configurations\ of\ children, \quad \frac{i.fairShare}{i.fairWeight} = \frac{j.fairShare}{j.fairWeight}$$<br>$$fairShare = \sum_{child} child.configuration.fairShare$$ |
| redistributeResources() | Fulfill preemption demands and regular demands of children in order. If available resources from parent queue are insufficient, some resources will be preempted from the children who can offer resources immediately. |

So with proposed scheduling framework, it is very easy to add a new scheduling policy. The developer just needs to define its own scheduling configurations and implement generic scheduling policy interface.

### Proposed Cluster Configurations

Fair scheduling is good at arbitrating resources between competing queues. Users (team or individual person) are the entities who can actively compete for resources. Inside each user, coordination rather than competition among applications is the main force. So the user should be the base unit of fair scheduling. Users can negotiable for a share of cluster resources initially. After that, each user is a self contained unit and can coordinate its own applications with specific scheduling policy which needn't to be fair scheduling.

Figure 5 is an example, At root level, fair scheduling is used to arbitrate resources among users. At lower levels, production users could use guaranteed capacity scheduling YARN-3807. Best effort user groups could use time based fair scheduling YARN-3808, each individual can use any scheduling policy wanted.

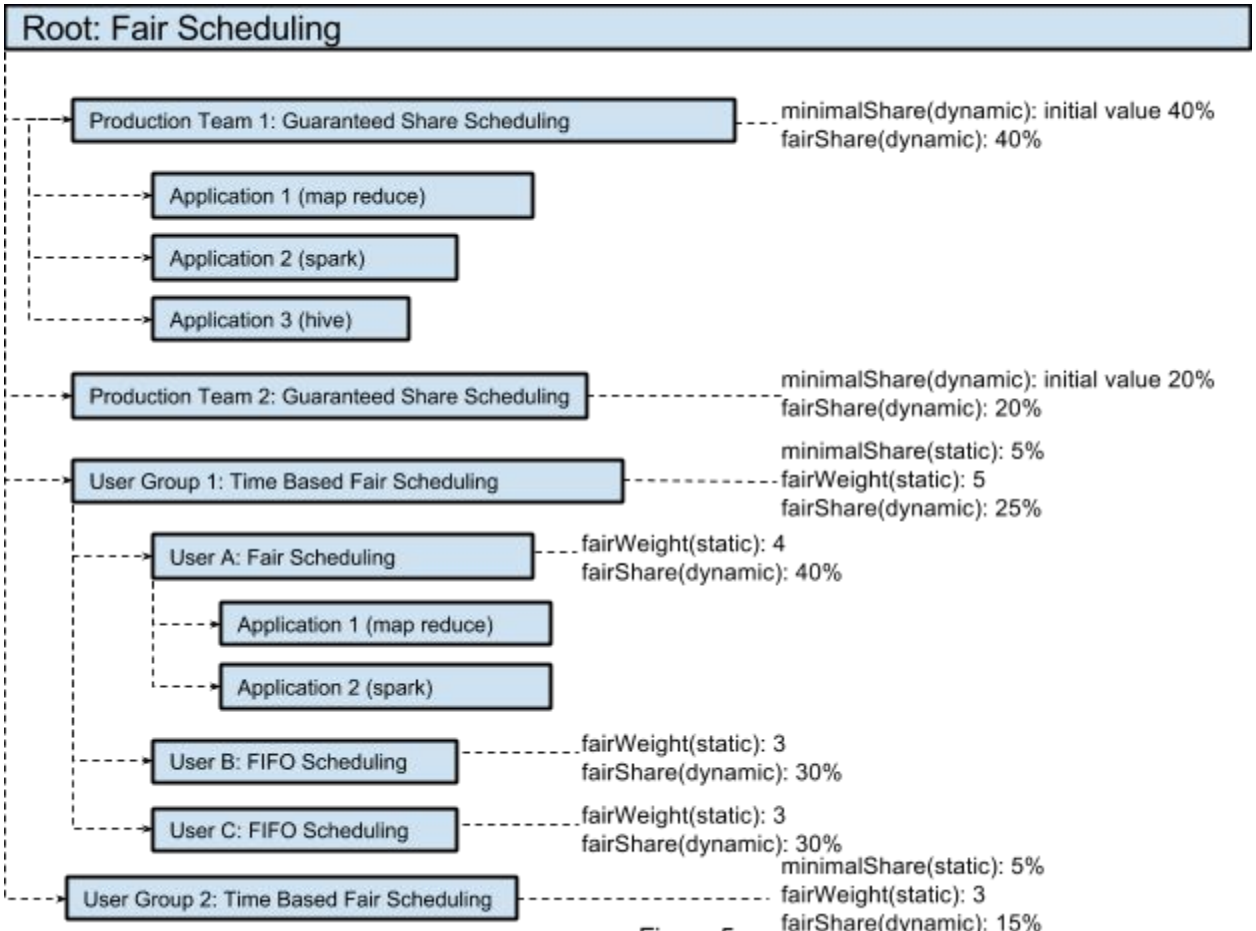

Figure 5

## Future Works

1. Detail design of ResourceManager and ApplicationManager.
2. Different scheduling policies share many common features. Libraries can be provided for new scheduling policy to implement generic scheduling policy interface easily.
3. Modeling cluster as a stochastic process and define metrics to measure scheduling performance.
4. Prototyping.

## Related Efforts

YARN-3807 Proposal of Guaranteed Capacity Scheduling for YARN
YARN-3808 Proposal of Time Based Fair Scheduling for YARN
YARN-3306 Proposing per-queue Policy driven scheduling in YARN
YARN-3405 FairScheduler's preemption cannot happen between sibling in some case