YARN-5139 (Global Scheduling) Design and Implementation notes

Wangda Tan
with input from
Vinod Kumar Vavilapalli, Karthik Kambatla and Daniel Templeton

Table of Contents

Problem statement

Existing YARN scheduling is based on node-heartbeats.

What this means is that we make scheduling decisions one node at a time - as each node comes in and heartbeats into the scheduler, the scheduler tries to pick the best application's best ResourceRequest that can use a container on this node.

Pseudo code of existing scheduling logic looks like:

```
for nodeHeartBeating:
  Go to parentQueue
    Go to leafQueue
      for application in leafQueue.applications:
        for resource-request in application.resource-requests
```

try to schedule on node

This one-node-at-a-time process can lead to suboptimal decisions. For example, for an application that prefers to run on one node in a one thousand nodes cluster, with the existing logic, the application will try to allocate on all the nodes one by one. It is possible that application gives up and gets allocated to a less-preferred node after too many tries.

Considering future complex resource placement requirements, such as node constraints (*give me "a && b || c"*) or anti-affinity ("*do not allocate HBase regionsevers and Storm workers on the same host"*), we need to consider moving YARN scheduler towards ***global scheduling***.

With global scheduling, YARN scheduler should be able to look at more nodes and select the best nodes based on application requirements unlike existing schedulers.

This can enable features like:

- Placement requirements like node constraints (YARN-3409), affinity/anti-affinity (YARN-1042)
- Find best nodes for requirements: For example, YARN-2745 (multiple resource packaging).

Requirements

The following is an attempt at coming up with (a) requirements from such a global scheduling mechanism and (b) open thoughts on what should be.

- Need to support both fast decisions and slow decisions
    - Today's scheduling path is primarily focused on response-time. It choses to spend the least amount of time making each scheduling decisions with a view to make as many decisions as possible in a short-time. This should be possible in the new structure too
- Nodes fail all the time. A global scheduling decision made should timeout if a node doesn't come back in time - this should be much less than node-expiry times. Leads to lots of "container-thrash" by the system - the app should-not be penalized
- Escape hatch for today's scheduling behaviour / behaviour compatibility ?
- Smooth rollout: Unlike preemption, we won't be able to roll this out -enable/disable per-queue. So, may we should have a way to switch between the scheduling behaviours at run-time?

Proposal

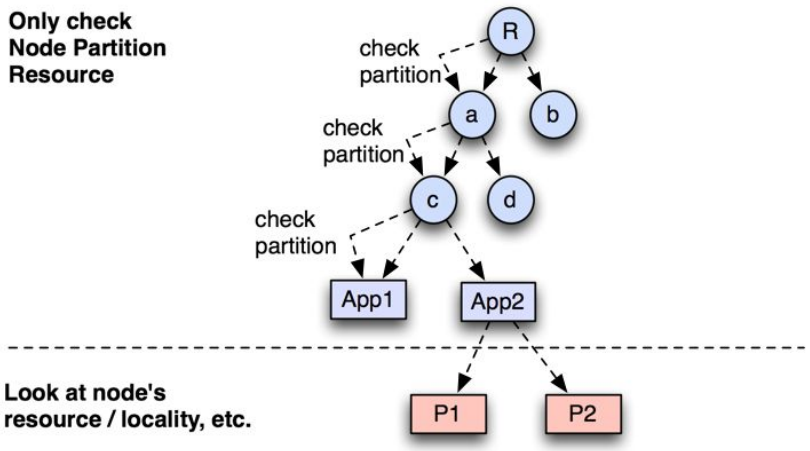
How existing scheduler works

For each node heartbeat (or each node visited by async scheduling), existing scheduler goes to root queue first, and then traverses to sub-queues and eventually to the applications in the leaf-queue based on limits, fairness etc.

For each application, the scheduler looks at resource-requests at different priorities and makes allocation decisions.

Please note that before scheduler goes to the resource-request level, *it only looks at general properties like total cluster resource, used resources, node-partition, etc. but nothing specific to the node heartbeating in.*

See the picture below, scheduler checks node specific properties of node only after reaching resource-request of P1/P2 (priority=1/2) of application-2 .



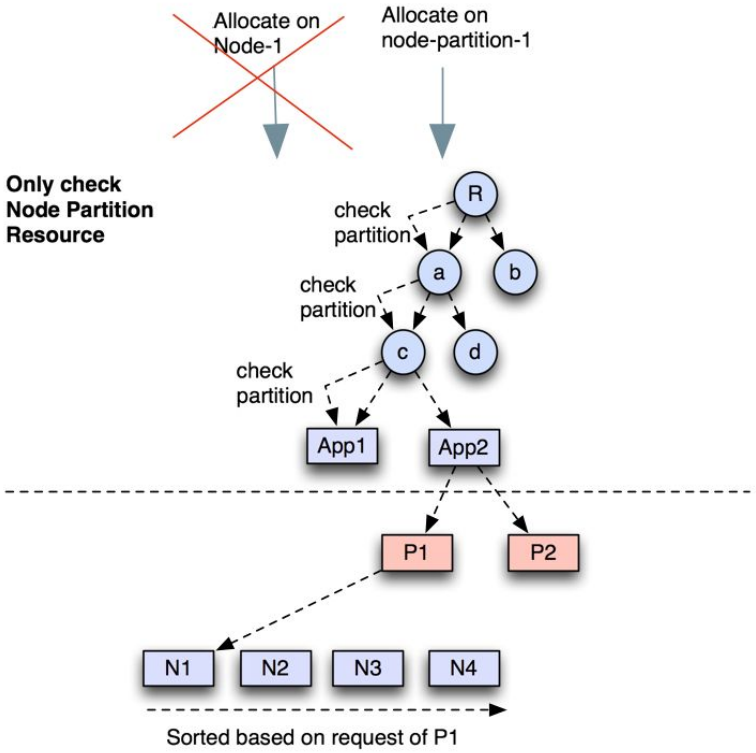
This characteristic is very important because we can consider which node to look at after we know which resource-request to allocate.

Proposed approach

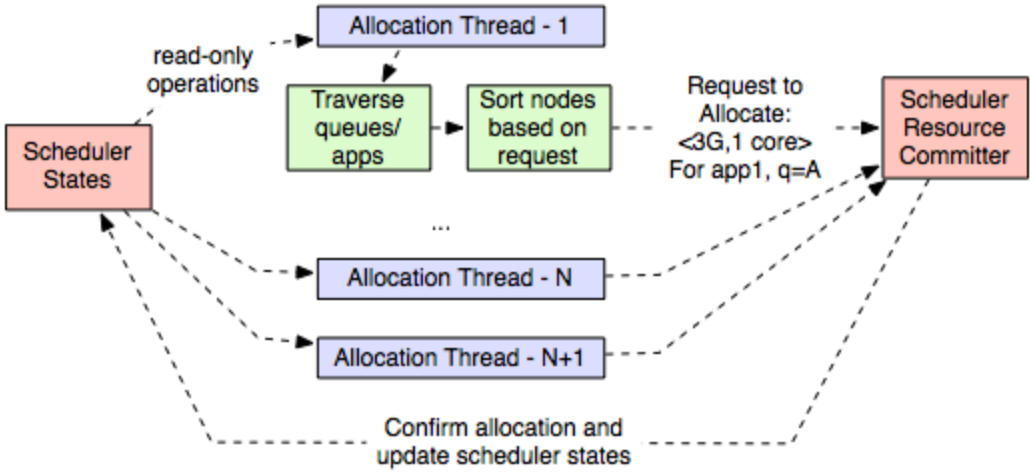
So, instead of only looking at one node at a time, we invert the process and late-bind the node-to-look-at after we figure out which resource-request is next.

See the picture below

- While doing allocation, we will first try to allocate on a node-partition instead of single node.
- After figuring out the next resource-request to allocate for, we can sort the list of cluster nodes based on the specific resource-request.



- To save the time spent on looking at the queue-structure, application-list and sorting nodes, we can parallel the above process:
    - Multiple threads can run in parallel to get the next resource-request and node, they will try to allocate on the node. If a new container can be either allocated or reserved on the node, it will send the allocation proposal (like allocate <3G mem> for application-2 from queue=A) to scheduler to allocate.

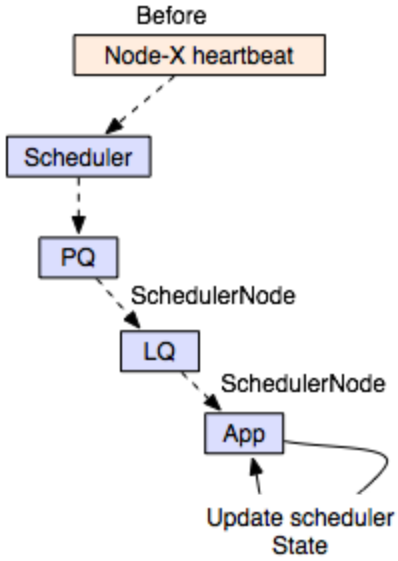


- Note that even in today's scheduling logic, for e.g in CapacityScheduler, we traverse the entire queue hierarchy for every single container-allocation. Even if in a single node-heartbeat we allocate multiple containers, each container still travels the full hierarchy. In that sense, moving to a ***get-candidate-container*** followed by ***commit-container-allocation*** will not add in terms of computation complexity.

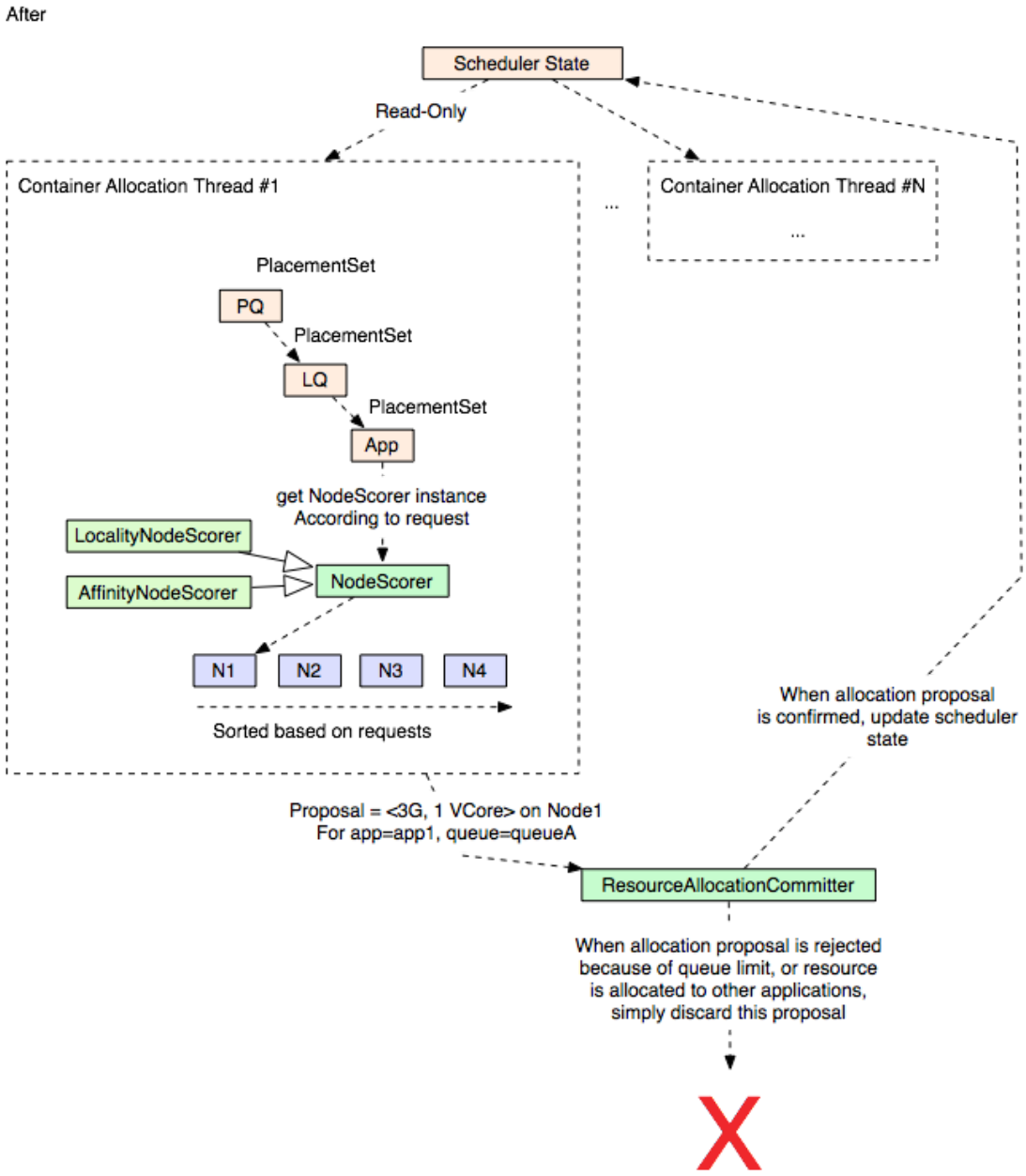
Implementation

Before this change

this global scheduling changes, scheduler selects SchedulerNode, pass down to ParentQueue (PQ), LeafQueue(LQ) and application.



After this change



Instead of passing down SchedulerNode, scheduler passes down a PlacementSet.

PlacementSet

It is a set of nodes preselected by scheduler, it has fields:

```
PlacementSet {
```

```
    Map<NodeId, SchedulerNode> nodeSet;
}
```

- *nodeSet* are preselected nodes by scheduler, application can choose from these nodes according to resource request. By default it is all nodes in the cluster.
- For today's applications which don't have powerful placement requirements, the scheduler can simply pass the next node heartbeating in as a singleton nodeSet - this is the key to make sure we don't need disruptive changes for existing apps.

NodeScorer

It is an interface to sort nodes according to resource request. It takes PlacementSet as input, and returns Iterator of SchedulerNode as output.

```
NodeScorer {
    Iterator<SchedulerNode> scorePlacementSet(PlacementSet candidates);
}
```

ResourceAllocationCommitter

This is an interface to merge allocation requests, it looks like:

```
public interface ResourceAllocationCommitter {
      void handle(ResourceAllocationCommitRequest request);
}

public class ResourceAllocationCommitRequest {
      List<AllocationProposal> containersToAllocate;
      List<ReservationProposal> containersToReserve;
      List<ContainerId> containersToRelease
}
```

Scheduler can implement the *ResourceAllocationCommitter* interface, and each *ResourceAllocationCommmitRequest* can include containers to be allocated / reserved / released.

A note about global optimal scheduling vs. this proposal

After the change, it still seems like instead of a full global view i.e. global view of all available resources and global view of all pending requests, we still seem to be doing one request at a time. You may ask “Won’t this lead to the same problems as today where a best fit is possible but the algorithm does not account for trying to do a global optimal allocation across all pending asks?”

Global optimal scheduling under our requirements is essentially an offline scheduler, considering we have so many scheduling constraints.

If you have read Choosy[1] paper before, you can see we have much more placement requirements than Choosy, for example, better packing / (anti-)affinity / user-limit. I would say we can only use offline scheduling to solve this problem unless we can prove this can be solved by an online algorithm in mathematics.

Introducing offline scheduler is unavoidably adding extra execution time, since we want to make YARN scheduler can support workloads that need fast scheduling, I would prefer to not support it at least for now.

Challenges of this approach

1) It seems like we need to fully sort the list of nodes for every resource-request, how do we reduce time spent on sorting?

Optimizations that we can do:

a. Use the parallel resource-allocator mentioned above to concurrently look at cluster state and find next container to allocate.
b. If an application isn’t too picky, for example, the app is requesting “*” host, we can include only one random node for the app to allocate.
c. Cache sorted result, for a lot of resource request, the order of preferred hosts will not be updated or can be in an incremental way. For example, order of preferred nodes for hard-node-locality request is fixed. And order of preferred nodes for anti-affinity is incremental (update once state changed).

Extra notes

- Having container-duration / deadline helps make global schedules a lot. “*I know that container is going to finish in a minute, and this app needs that node ..*”

References

[1] Choosy: Max-Min Fair Sharing for Datacenter Jobs with Constraints, https://people.csail.mit.edu/matei/papers/2013/eurosys_choosy.pdf

[2] Borg: Large-scale cluster management at Google with Borg, http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43438.pdf

[3] Large-scale cluster management at Google with BorgQuincy: Fair Scheduling for Distributed Computing Clusters, http://www.sigops.org/sosp/sosp09/papers/isard-sosp09.pdf