FairScheduler Preemption Overhaul

As documented on YARN-4752, there have been a number of issues reported around preemption when using FairScheduler. FairScheduler's preemption was implemented a while ago, even before it supported hierarchical queues and pluggable policies. Coming from the MR1 world, most Yarn clusters still had mostly same-sized containers. Today, the queue configurations are more complex and the varied workload resource requirements are reflected in the resource requests.

The community has raised a bunch of issues and proposed solutions to fix these issues within the current scheme of things. This document is an attempt to look at these issues holistically.

Current issues with preemption

Prioritized list of current issues:

1. FS uses a spray-gun approach for preemption. It preempts resources from multiple nodes - the sum of preempted resources might be enough to fit a new container, but any one node might not have enough resources. (YARN-2154)
2. Doesn't consider preempting resources from sibling queues. (YARN-3405)
3. Doesn't consider application starvation. (YARN-4333, YARN-3997)
4. Resource accounting can be dicey:
    a. a queue's resource usage includes resources slated for preemption. (YARN-4120)
    b. Containers can escape preemption even after being tagged so. (YARN-4133)
5. AM and non-AM containers are treated the same. (YARN-3902)
6. Insight into the extent of preemption going on in the cluster. (YARN-3121)

There are more long-term feature/improvement requests, that are likely better left out of this exercise.

Prioritization

The primary motivation for preemption is to avoid fairshare starvation of queues and applications, while maintaining high utilization when high-priority queues have no active workloads. While issues (2) and (3) are to do with queues/applications starving, issue (1) preempts containers that don't even meet the requirements of a starving queue/application.

Consolidated Approach

Let us approach the issues in the prioritized order. Solving (1) correctly addresses (2), (3) and (5) as well. (4) is bugs in current implementation; we want to add tests to ensure the new

implementation doesn't have the same issues. (6) is nice-to-have and allows us to investigate tradeoffs.

The current approach is to identify the amount of resources required by starved applications and preempt those resources from across the cluster. Consider the case where only one application/queue is starving, and the starved application requests 4GB containers. With the current approach, it is possible for the scheduler to preempt 4 1GB containers from 4 different nodes.

Clearly, the scheduler needs to

- Identify starved applications.
- Use the actual resource requests of starved applications instead of cumulative amount of resources to be preempted.
- For each resource request, identify one node with enough free and/or preemptable resources to meet the requirement.

Identifying starved applications

1. Since the starvation is in relation to a queue/application's fairshare, it seems reasonable to check for starvation when we update the fairshare of a *Schedulable* in the update thread.
2. In addition to finding the amount of starvation, we should also track the starved applications.
3. It might be prudent to maintain a global *PrioritizedQueue* of *starvedApps,* where applications starved by larger amounts and/or for longer durations are prioritized.

Using ResourceRequests (RRs) of starved applications

1. The outstanding RRs are tracked in *FSAppAttempt* (via *SchedulerApplicationAttempt*).
2. We want to consider one RR at a time to satisfy, and return the application to the prioritized list of starved apps if it is still starved. This way, we don't starve other starved applications (pun intended :-)).
3. We could pick RRs in the order of RR-priority, followed by the degree of locality relaxation - off-switch, rack-local-only, node-local-only. Note the -only: we don't want to start preempting resources even before the RR is fully relaxed (goes through delay scheduling).

Identifying the right node

1. Based on the allowed locality levels of a RR, we should determine the set of candidate nodes.
2. For each candidate node, we check if it is running enough containers from applications over their fairshare to meet the RR in question. This also considers the unallocated resources on the node.

3. If we do find enough resources to accommodate the incoming request, we go ahead and reserve those resources, so smaller containers don't sneak in as we preempt the identified containers one by one.

Implementation and Testing

Since all the work is tightly coupled and partial implementation is non-functional, using a branch would allow for multiple easy-to-review small commits.

For testing, we should adapt existing unit tests to the new implementation and add additional tests to verify issues (1) through (5) are fixed.