[Design] Proposing per-queue Policy driven scheduling in YARN

Craig Welch, Jian He, Wangda Tan, Vinod Kumar Vavilapalli

Last modified: March 9 2015

Scheduler development in Apache Hadoop YARN today is very coarse grained and insular. This proposal aims at converting today's rigid scheduling in YARN to a per-queue policy driven architecture.

Present state of scheduling

YARN today has a pluggable top level ***ResourceScheduler*** sitting inside the ResourceManager daemon and interacting with the rest of the ResourceManager (RM) components. It is responsible for allocating free resources on the managed nodes in a cluster to applications. It has the following characteristics:

- At any point of time, *only one scheduler* can be configured and is in play in a ResourceManager
- Applications are organized into a *hierarchy of queues* that each scheduler is responsible for.
- The scheduler is responsible for *all the scheduling decisions* happening in *all queues* and applications at all levels
- Each scheduler has to handle incoming events from the rest of the ResourceManager components such as node addition/removal, application start/finish, application-attempt start/finish, container release, allocate calls from ApplicationMasters (AMs) and node-heartbeats.
- Handling node-heartbeats forms the core scheduling cycle.

The following are common to all the schedulers

- The notion of a *Queue*, each *Queue's Metrics*, *Queue level ACLs* to control who can submit applications and who can administer queues
- The notion of a *SchedulerApplication*, that includes basic information corresponding to each application, irrespective of the mechanism of how resources are distributed
- The notion of a *SchedulerApplicationAttempt* that includes information corresponding to each ApplicationAttempt. Some of this information is used by both the schedulers, but some of it isn't.

We have three concrete implementations of these coarse grained schedulers supported by the YARN community: Capacity Scheduler, Fair Scheduler and a Fifo Scheduler which is mostly used in tests.

Problems with the current state

The way the scheduling is laid out in YARN, there are multiple problems:

Rigidity

Scheduling today is very rigid. Implementing a new policy essentially amounts to either implementing a new scheduler or changing one of the existing schedulers also to shoehorn the new policy.

Why not implement multiple *ResourceSchedulers* to achieve different behaviors? YARN's *ResourceScheduler* interface is a high level abstraction which encompasses a very large feature set and functionality of the ResourceManager. It's an appropriate abstraction for high level variations but is awkward to use for modifying fine-grained behaviors like container assignment and preemption. The ability to mix and match particular pieces of Scheduler behavior calls for focused interfaces for particular operations (Queue/Application Ordering for Container Assignment and Preemption; Queue, User, and Application Limits, etc) which can be composited within (and definitely shared across) specific ResourceScheduler implementations.

Modifying existing schedulers for each new policy is possible in theory, but in practice only works as long as we do not have to implement two conflicting algorithms in the same Queue. Enabling multiple conflicting algorithms can still be achieved by making use of configuration knobs, but that usually becomes awkward and error-prone in a very short order.

Difficulty of experimentation

Every scheduler's LeafQueue implements its own algorithm from scratch. So, in order to be able to even do small incremental enhancements, there is no choice but to go change existing schedulers and risk the stability of existing algorithms.

Further, even though a cluster administrator may like to experiment with a newly written algorithm only on a subset of the cluster's queues, he/she cannot do it as things stand today. We've seen many cases of this, the most recent one with the roll out of preemption - YARN-2056 - we've addressed that problem by adding one more configuration knob to be able to control preemption per queue, but clearly this doesn't scale.

Fragmentation

Today's scheduling is fragmented in YARN. Different schedulers implement their own non-unified features. Further, even though there are many internal components (for e.g Application/Queue ACLs management) that are common to all schedulers, the way they are used is left to the scheduler in question. Many a time, this leads to functionality being implemented first only in one scheduler but not in others till a later point of time, leading to a broken user/application experience. Node blacklisting/whitelisting APIs, ability to move applications across queues, reservations are some such examples.

Though this is a long list of problems to address, in this document we limit ourselves to solving the rigidity/experimentation problem with the introduction of per-queue policies which should help the algorithm reuse and extensibility.

Proposal

We propose the creation of a *common policy framework* and implement *a common set of policies* that administrators can pick and chose per leaf-queue

- Make scheduling policies configurable per queue
- Initially, we limit ourselves to a new type of scheduling policy that determines the ordering of applications within the *leaf-queue*
- In the near future, we will also pursue parent-queue level policies and potential algorithm reuse through a separate type of policies that control resource limits per queue, user, application etc.

So, the new LeafQueue will look like the following

```
assignResources() {
        while (stillMoreResourcesToShare()) {
          for (Application a : appsAsOrderedByPolicy) {
            // Figure out the limit to apply, this can be a separate policy type later
           assignResources(a, totalFreeResouce, limit);
          }
        }
}
```

In this document, we are not writing down the specific policy interface as it goes through the evolution, but the chief characteristics of a leaf-queue policy are below

- Each leaf-queue's ordering-policy controls the order in which applications are considered for resource allocation and preemption.
- Examples of ordering-policy implementations
    - Priority: To support user-facing Application Priorities
    - FIFO, LIFO
    - CS Policy: Today's Capacity Scheduler with user-limits etc
    - Fair-sharing: Today's Fair Scheduler's policy
- Ordering policies can be applied hierarchically where it makes sense. For example, the Priority policy can be combined with other policies, typically, as a parent to establish the desired ordering *within* priority bands
- The Scheduler's limits (or limit-policies if/when implemented) constrain resources which can be provided to an application.

We will propose finer details of the policy interface in the main sub-task for further discussion.

Implementation details

Overall, we plan to have the following in this implementation:

- We first work on a common policy framework with a common-set of policies available in common packages.
- Then, we work on specifically using them separately in Capacity Scheduler followed by the Fair Scheduler. If at all, there is a possibility for more convergence (beyond policies) between the schedulers, it will be pursued later independently.
- Initially (and for the near future), the policy-framework will be limited to being private and the list of supported leaf-queue policies will be kept small and focused. This reduces surface area which can result in tight coupling with individual schedulers. It also makes reuse simple by minimizing the code required for incorporation and dependencies necessary for use.
- The policies will be located in common source packages using classes common across schedulers and available for use in any scheduler, and the CapacityScheduler will first be modified to use them. Default behavior for the CapacityScheduler will be kept backward-compatible with present behavior.

Note that Fair Scheduler already has a notion of policies applicable to leaf and parent queue. Given our goal of a common policy framework and policy implementations universally applicable and reusable, we are considering as much code/concept reuse as possible with the existing functionality. In order to provide for maximal reusability and composability, the

existing logic will be recast to a different factoring for initial use in the CapacityScheduler and possible sharing between schedulers down the road.

Configuration & Management Changes

Leaf queues shall have additional optional configuration points to be able to specify ordering-policies to be applied. Configuration shall support short names (fifo, fair) and potentially also accept any classname which implements the necessary interface. A parallel effort [1] to unify schedulers' configuration should be of help. Further, APIs and user interfaces which provide information about each queue shall be enhanced to provide or display the active policies.

Extensions

As hinted before, we will extend this policy framework down the road to also apply to distribution of resources by the parent-queue to its children. Further, we are also considering extracting the limits per queue, user, application in the form of composable, reusable policies.

Related efforts

1. YARN-2986 (Umbrella) Support hierarchical and unified scheduler configuration