July
2016

Scheduling of Long-Running Applications (LRA) in YARN

KONSTANTINOS KARANASOS, ARUN SURESH, PANAGIOTIS GAREFALAKIS, SARVESH SAKALANAGA, CARLO CURINO, KISHORE CHALIPARAMBIL, SRIRAM RAO

Table of Contents

1. Motivation

The current schedulers in YARN (namely, the Capacity and the Fair Scheduler), when placing tasks, take into account the available resources of each node, various sharing constraints (such as capacity and fairness), as well as some "static" locality preferences. In particular, an application can specify *data locality* constraints, that is, at submission time, it can determine whether a given task should be placed at a specific node or rack.

More recently, support for *node labels* was also added to YARN. Each Node Manager can specify a node label (e.g., to advertise that it is equipped with a GPU or an FPGA). Applications can subsequently request their tasks to be placed on nodes with specific node labels. Although node labels increased the flexibility of the schedulers, they still correspond to *static* constraints, in the sense that node labels are assigned when NMs start their execution and are not expected to change. Thus, they do not reflect the condition of the cluster at a given time (e.g., to show that an application is already using the GPU at a node in order to avoid placing a second task in the same node).

The already supported constraints are mostly sufficient for scheduling analytics jobs, which was the initial target of YARN. However, there is an increasing demand for scheduling applications comprising long-running tasks (such as services or executors for interactive queries), alongside existing analytics jobs. We term such applications Long Running Application or LRA, for short. This is an initiative that started with Slider, and currently more native support for services is being added to YARN (see YARN-4692). Carefully scheduling LRAs would require a richer set of constraints to be supported by the scheduler. Indeed, given that LRAs are expected to run for extended periods, suboptimal placement decisions would have a bigger impact in the system's performance. Furthermore, many LRAs (e.g., HBase or Heron), are more prone to latency problems, compared to analytics jobs.

In this document we describe how we extend YARN's scheduler in order to support the placement of long-running jobs. First, we give an overview of our design, then we give more details on how LRAs can determine their scheduling preferences in the form of placement and time constraints, and finally we present how the scheduling and planning algorithm can be extended to take into account the richer set of constraints.

2. Design Overview

In this section, we first provide an overview of the system's architecture, which is depicted in Figure 1. As can be seen from the figure, we have introduced two new components in the YARN RM, namely the LRA Planner and the Constraint Manager. Moreover, we have extended the Scheduler and the Common Label Manager.

When a client submits a service, it determines a set of tags for each of its tasks and a set of constraints. These dynamic tags are handled by the Common Label Manager (more details about dynamic tags are given in Section 3.1), whereas the constraints are handled by the Constraint Manager (Section 3.2).

At submission time, the user can decide whether the LRA will be treated by the scheduler (Section 4.1), which is faster but leads to sub-optimal allocations, or through the planner (Section 4.2) that can perform more optimal placement of LRAs and supports a richer set of constraints.

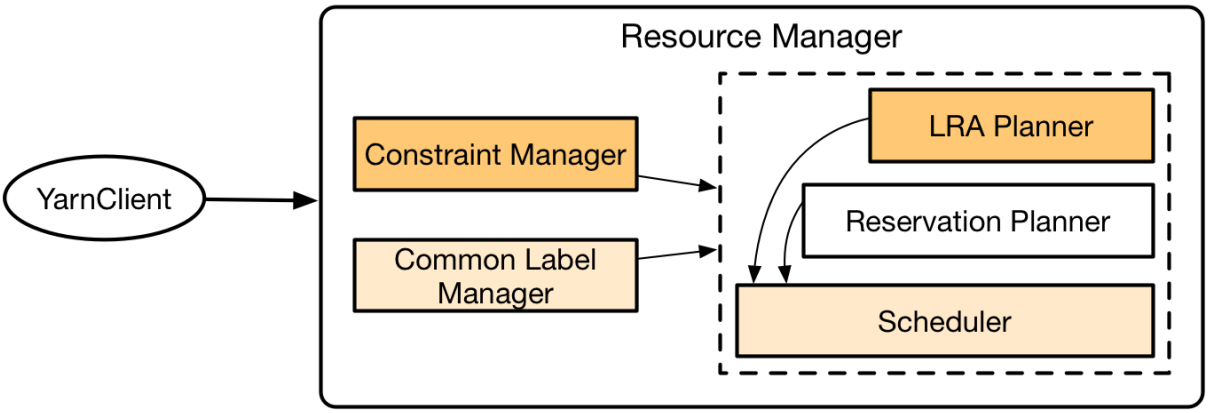

Figure 1. Architecture

3. Defining LRAs: Dynamic Tags and Constraints

Each LRA comprises a set of Resource Requests (augmented with a set of dynamic tags), along with a set of placement and time constraints. Below we fist discuss the dynamic tags (Section 3.1) and then present the constraints (Section 3.2).

3.1 Dynamic Tags

Dynamic tags are similar to node labels with the difference that they can change over time, as new jobs enter the system and others finish their execution. Supporting dynamic tags is crucial for the scheduling of LRAs. Note that a subset of these tags can also be statically defined (e.g., to identify nodes with GPUs or FPGAs). Here we describe how we extend the Common Label Manager to support dynamic tags.

Task, Node and Rack Tags

We add a new field in the Resource Request to specify the set of tags that are associated with the corresponding tasks. Then, when a task gets allocated to a node by the scheduler, the set of tags of that task get added to the node. When the task finishes its execution, the associated tags get removed from the node as well.
Hence, a node inherits the tags of the tasks that are currently allocated to it. Likewise, a rack inherits the tags of its nodes.

For instance, consider an HBase Master task with tag `hbase-m` that gets allocated in node n1 of rack r1, and an HBase RegionServer task with tag `hbase-rs` that gets allocated in node n2 of r1. Node n1 has an `hbase-m` tag, n2 has an `hbase-rs` tag, whereas rack r1 has the set of tags {`hbase-m`, `hbase-rs`}.

Node partitions

We introduce the notion of node partitions to define a set of nodes, which can then be referred to in placement constraints, as we discuss in the next section. A rack is a specific case of a node partition, containing all nodes of a rack. Another example of a node partition is an upgrade domain, used to

define a set of machines that are upgraded at the same time in a cluster. Similar to the RackResolver, there exists a NodePartitionResolver for each node partition in order to map nodes to node partitions. Note that each node partition inherits the tags of the nodes that belong to it (tantamount to the rack tags). These are the *node partition tags*.

3.2 Placement and Time Constraints

The placement constraints specify constraints that need to be satisfied when placing the tasks of the LRA to the cluster nodes (such as "don't put more than four tasks of this LRA at the same rack"). The time constraints determine for how long a set of resource requests and placement constraints are valid (such as "place these 10 Heron containers with some placement constraints from 8am to 2pm, and place 20 Heron containers with some other placement constraints from 2pm to 10pm").

Placement Constraints

We define a placement constraint as follows:

$$C_p = \{source,\ target,\ scope,\ min\_cardinality,\ max\_cardinality\},$$

where *source* and *target* are dynamic tags. Note that both the source and the target can refer to a single task by using its task ID as the allocation tag. Moreover, the source and target can coincide, in order to specify constraints that refer to a specific group of nodes.
Scope is a node partition, whereas *min_cardinality* and *max_cardinality* are integers. A value of -1 signifies that there is no constraint on respective cardinality.

The semantics of a placement constraint is that tasks annotated with the *source* tag can be placed along with tasks annotated with the *target* tag, as long as there are no less than *min_cardinality* such tasks and no more than *max_cardinality* such tasks within the node partition used as the *scope* of the constraint.

Soft and hard cardinality limits

When defining a constraint, the AM can specify whether min_ and max_cardinality are soft or hard limits. We do so by associating an integer with each of the limits, denoting how long the AM is willing to wait for its constraint to be satisfied. A small value indicates that a constraint is soft, whereas a MAX_INT value indicates a hard constraint.

Combining multiple constraints

Multiple constraints can be combined with AND and OR operators, forming more complex constraints. Consider the following example:

We allow complex constraints to be expressed in disjunctive normal form (DNF), which allows us to express any combination of constraints.

Intra- and inter-application constraints

Placement constraints may refer either to tasks within the same LRA (intra-application) or across applications.

Clearly, intra-application constraints are easy to define, given that the application has visibility of all its resource requests. Inter-application constraints can be more complicated to define. One way of defining such constraints is by using the same generic dynamic tag across applications (e.g., not allow more than two HBase masters at the same node). However, for a job to refer directly to the tasks of another specific job, we need to use its application ID, which in turn means that the

application has to already be deployed (and assuming there is a way to learn the ID of that application). In Section 4.2 we explain how we can work around this issue.

Application and global constraints

So far we have assumed that constraints are only specified by the AM when submitting an LRA. Along with such constraints, the cluster operator can publish its own constraints that act as global constraints. These can be added directly to the Constraint Manager of the RM.

For instance, the cluster operator can impose a constraint of the form {“am”, “am”, rack, -1, 10}, which means that it is not allowed to have more than 10 AM containers running at each rack.

Time Constraints

We define a time constraint as follows:

$$C_t = \Big\{\{RR\},\ \{\mathrm{C}_p\},\ start\_time,\ end\_time\Big\},$$

where {RR} is a set of resource requests, $\{C_p\}$ a set of placement constraints and [start_time, end_time) defines the interval in which these constraints will be valid.

4. Scheduling and Planning of LRAs

4.1 LRA Scheduling

When an LRA gets submitted to the RM through the YarnClient, the constraints for this application are added to the ConstraintManager. Then the scheduler tries to allocate the given resource requests, along with the placement constraints, in the following way:

for each node *n*
  for each application *app*
    for each resource request of *app*
      if all constraints of *app* are satisfied in *n*
        try to schedule on *n*

Note that using the global scheduling as proposed in [YARN-5139], where we do not iterate over the nodes as they heartbeat, but rather select the most promising nodes, would result to even better scheduling decisions.

Shortcoming of LRA scheduling

Although the placement of LRAs through the existing scheduling infrastructure, as described above, can accommodate multiple use cases of placing LRAs, it has the following limitations:

- The scheduler considers only a single resource request at a time and has no holistic view of the other services that need to be scheduled (or even or the other resource requests within the same service). Hence, it can miss more optimal placement opportunities.
- As explained above, support for inter-application constraints is limited with the scheduling. Often times it might be required to have some applications already deployed and know their application ID in order to refer to their tasks in another application’s constraints.
- For similar reasons, time constraints are hard to be imposed.

To address these drawbacks, we are introducing the notion of LRA planning.

4.2 LRA Planning

Instead of going directly to the scheduler, a user can submit its LRA through the LRA planner.

The LRA planner, instead of looking at a single resource request at a time, takes into account multiple services at once. This way, more involved techniques, such as linear programming, can be used to place the LRAs. After each iteration, the planner outputs the best possible allocation for the tasks of the LRAs to the cluster nodes. Then, a set of updated resource requests are generated for each LRA. These resource requests specify the exact nodes to be placed, following the output of the planner.

These updated resource requests are submitted to the scheduler that is responsible for doing the actual allocation. Notice that the condition of the cluster between the moment the planning happened and the moment the scheduler allocated those containers might have changed. To this end, the updated resource requests are still augmented with the placement constraints. If the cluster condition has indeed changed, the scheduler will take the constraints into account and perform dynamic micro-adjustments to the planner's allocation. We are also investigating alternative ways of performing these runtime adjustments.

The LRA planner can also consider migrating containers that have already been allocated in a previous iteration of the planner, if that would lead to a better allocation, also taking into account the cost of migration.

The fact that a user can submit multiple LRAs at the same time, allows users to easily specify detailed inter-application constraints.