March 2017

[YARN-5468] Scheduling of Long-Running Applications

KONSTANTINOS KARANASOS, ARUN SURESH, PANAGIOTIS GAREFALAKIS,
SUBRU KRISHNAN, CHRIS DOUGLAS, CARLO CURINO, SRIRAM RAO

Table of Contents

1. Motivation 1
2. LRA scenario 2
3. API for placement constraints 2
3.1 Allocation tags 2
3.2 Node groups 3
3.3 Placement constraints 4
4. Scheduling an LRA 6

1. Motivation

The focus of YARN so far has been mostly on traditional analytics jobs, each consisting of relatively short-running tasks. However, we observe an increasing demand for applications using long-running containers, alongside existing analytics jobs. Such tasks might run for hours, days or even months. We term jobs falling in this category **long-running applications** (or LRAs).

There has been a significant effort lately to extend YARN to support the needs of LRAs, e.g., YARN-4692, YARN-4793, YARN-5079.

In this document, we focus on the scheduling aspect of LRAs. Careful placement of long-running containers is key to the *performance* and *resilience* of LRAs. For example, it may be beneficial to co-locate applications' containers on the same rack (**affinity**) to reduce network costs, spread containers across machines (**anti-affinity**) to minimize resource interference, or allow up to a specific number of containers in a node group (**cardinality**) to strike a balance between the two. Placement decisions also affect resilience. For example, containers placed within the same cluster upgrade domain would go offline simultaneously. Note that LRAs rely on invariants *between* nodes and not *of* specific nodes (unlike, for example, node labels). More details are given in Section 3.

At the same time, the cluster operator should be able to place LRAs in a way that balances load, reduces resource fragmentation, and puts limits to the constraints the applications can impose.

We take the view that LRAs should be scheduled outside the main scheduler's loop, since they have different scheduling requirements. To this end, we **intercept all LRA traffic**, generate synthetic requests and then feed them to the main scheduler (see Section 4).

To sum up, our **requirements** when scheduling LRAs are the following:

R1. Support expressive constraints, such as affinity, anti-affinity and cardinality, both within and across apps, without requiring the application to know the underlying topology of the cluster or the other applications deployed;
R2. Holistically place long-running containers considering multiple resource requests and constraints at a time;
R3. Allow the cluster operator to optimize for various objectives (e.g., balance load across nodes or minimize resource fragmentation) and put limits on the LRAs;
R4. Be external to Capacity/Fair scheduler, so that it can be used by any scheduler without affecting the scheduling latency of batch analytics applications.

2. LRA scenario

Figure 1 shows a toy cluster consisting of eight nodes in two racks, split across two upgrade domains (i.e., set of nodes upgraded together).

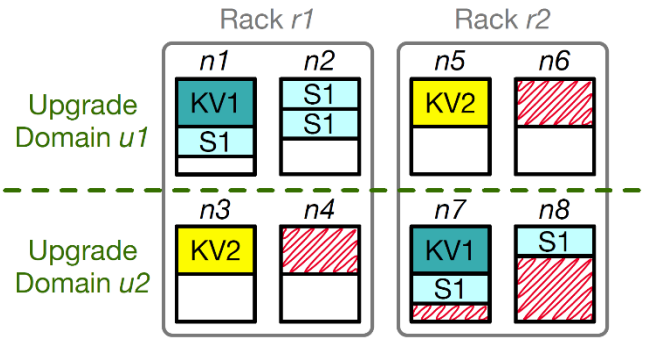

Figure 1 LRA scenario with key/value store instances (KV1, KV2) and stream job (S1).

In this cluster, two instances of a distributed key/value store (KV), such as HBase, are deployed, each using two containers (KV1, KV2). A streaming job, S1, consuming data from KV1, is also running in the cluster. Used resources are marked with red.

In this scenario, each container of the key/value store instances should be placed across different fault (rack) and upgrade domains, to guarantee the resilience of the applications. To reduce network traffic and improve performance, two of the S1 containers should be placed on the same rack, and all should be placed in the same rack as the KV1 container from which they consume data. Moreover, S1 might request to not place more than three of its containers on the same rack, to not overload the network and for fault tolerance.

3. API for placement constraints

In this section, we discuss the API for the LRAs to define their placement constraints. Before defining the constraint API, we introduce the notions of allocation tags and node groups that are used in the constraints.

3.1 Allocation tags

Each container of an LRA can be associated with a set of string tags. For example, the containers of KV1 in our example of Figure 1, can be associated with tags `KV` to show the type of application they belong to, `latency-critical` to refer to the more general demands of the container, `app_0041` to denote to the application ID.

An important use of tags is to identify *components* of LRAs, as also required in YARN-4793. For example, an HBase Master container can be tagged with `hbase-m`, and Region Servers with `hbase-rs`.

Tags are a simple yet powerful mechanism for constraints to refer to multiple containers, belonging to the same or different applications. As we show later, a constraint can use tag `hbase-rs` to refer to any HBase Region Server container.

Specifying allocation tags

Allocation tags are specified in the **AllocateRequest** object. We add a map between the ContainerRequest and a set of string tags. The mapping between the ContainerRequest and the list

of tags is done through the AllocationRequestID. An alternative design would be to add the tag set directly in the ResourceRequest, but we decided to not add more fields in the already loaded ResourceRequest object.

As an example, consider an AllocateRequest for one HBase Master container and two HBase Region Servers. All containers get an **hbase** tag, as well as an **hbase-m** or **hbase-rs** tag, depending on the role:

```
AllocateRequest {
    ResourceRequest { ..., numContainers=1, allocationRequestID=23 },
    ResourceRequest { ..., numContainers=3, allocationRequestID=24 },
    AllocationTagsMap { { 23, { hbase, hbase-m } }, { 24, { hbase, hbase-rs } }
}
```

Difference with node labels and node attributes

The difference between allocation tags and existing node labels or node attributes (YARN-3409), is that allocation tags are attached to containers and not to nodes. Moreover, similar to node labels and unlike node attributes, allocation tags have no value attached to them. Allocation tags are also considered in the YARN-4902 proposal, although in that JIRA they have the form of key-value pairs.

Note that in our constraint expressions we will show that we allow specifying constraints that include node labels, node attributes, as well as allocation tags.

Allocation tag sets of nodes and racks

When a container gets allocated to a node by the scheduler, the set of tags of that container gets added to the node. When the container finishes its execution, the associated tags get removed from the node too. Hence, a node inherits the tags of the containers that are currently allocated to it. Likewise, a rack inherits the tags of its nodes.

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

Allocation tags should be stored carefully in in-memory data structures, which can be updated and queried fast without taking up much memory.

3.2 Node groups

We introduce the notion of node groups to define categories for sets of nodes. The simplest, predefined node groups are `node` and `rack`. A node set belonging to the node group `node` includes a single element that corresponds to a cluster node. A `rack` node set contains all nodes of a physical rack. In Figure 1, node sets {n1}, {n2}, …, {n8} belong to node group `node`, whereas {n1, n2, n3, n4} and {n5, n6, n7, n8} belong to node group `rack`.

Apart from the predefined `node` and `rack` node groups, we allow cluster operators to register and update custom node groups. Such examples are `fault domains` (i.e., hosts that are more likely to fail together) and `upgrade domains` (i.e., hosts whose bits get upgraded simultaneously), or nodes connected with each other through InfiniBand. Node groups enable constraints to be expressed independently of the cluster's underlying organization. For example, a constraint requiring to "place

`hbase` containers of the same application in different upgrade domains", does not need to be aware of the cluster's upgrade domains or perform any actions when upgrade domains change.

3.3 Placement constraints

We allow applications to define the following constraints:

- `affinity(container-group, target-group, node-group)`: each of the containers having an allocation tag `container-group` should be placed within the same `node-group` with a container having an allocation tag `target-group`.
- `anti-affinity(container-group, target-group, node-group)`: each of the containers having an allocation tag `container-group` should not be placed in the same `node-group` with any container having an allocation tag `target-group`.
- `max-cardinality(container-group, target-group, node-group, c_max)`: each of the containers having an allocation tag `container-group` should not be placed within the same `node-group` with more than `c_max` containers having an allocation tag `target-group`.

Examples

Below we give various examples of our constraints:

```
// Place HBase Region Servers on the same rack
affinity(hbase-rs, hbase-rs, rack)

// Place HBase Masters in different nodes
anti-affinity(hbase-m, hbase-m, node)

// Place no more than 3 ZooKeeper instances on the same rack
max-cardinality(zk, zk, rack, 3)

// Collocate Storm containers of app 0045 with an HBase Region Server of app 0089
affinity(storm AND app_0045, hbase-rs AND app_0089, node)

// Spread Storm containers across upgrade domains
anti-affinity(storm, storm, upgrade_domain)
```

Note that the above constraints can naturally capture the affinity and anti-affinity constraints mentioned in YARN-4793 too.

Specifying constraints

Placement constraints are specified as a string either in the **ApplicationSubmissionContext** or in an **AllocateRequest**.

LRA-wide constraints are specified in the ApplicationSubmissionContext and should be respected by all AllocateRequests of the LRA.

To capture the cases when a new AllocateRequest arrives and needs to add some specific constraints, we also allow constraints to be added at an AllocateRequest. These constraints add to the LRA-wide constraints, but their scope is restricted to the containers of the AllocateRequest.

However, in most cases, we expect the constraint expression at the ApplicationSubmissionContext to be sufficient. This is the case in all our internal applications. And this is also the case in the scenarios described in YARN-4793, where all constraints (called PlacementStrategies in that JIRA) are specified in the JSON file that is given at LRA submission. The constraints in YARN-4793 are either LRA-wide or component-wide, both of which can be specified in the ApplicationSubmissionContext.

Why not add constraint expressions at the ResourceRequests

We considered adding constraint expressions to the ResourceRequest level too, but decided not to do so for the following reasons:

- Constraints always capture interactions between containers, and do not refer to a single container (which would justify adding them in the ResourceRequest). We found no use cases where that would be useful.
- Specifying the constraints in ApplicationSubmissionContext or AllocateRequest allows us to group constraints and be more succinct. In an application with 1000 ResourceRequests within an AllocateRequest, we wouldn't want to alter all 1000 requests to include constraints that can be captured by a single constraint in a higher level.
- Not adding them in the ResourceRequest level also facilitates storing the constraints at the RM and allows to easily update them by the cluster operator.

String representation of constraint expressions

We use a string representation of a constraint expression to avoid all the *PBImpl classes, especially given that the API might change quite a lot based on discussions with the community. Having a string value will allow changes in the API with less extensive code changes. A validation of the string syntax can be added in the client side to avoid user errors in specifying constraints.

Moreover, to allow our placement constraints to be combined with node label and node attribute expressions (see YARN-3409), we use **namespaces** in our string expression, as shown in the following example:

```
node-label:blue AND node-attribute:GPU=NVIDIA AND affinity(hbase-rs,
hbase-rs, rack)
```

Common API for LRA constraints

Note that affinity and anti-affinity constraints (but not cardinality constraints) are considered in YARN-1042 too. In that JIRA, the discussed implementation in YARN-4902 focuses only on specifying constraints at the ResourceRequest level and considers only the Capacity Scheduler.

YARN would not benefit from multiple, incompatible constraint APIs for LRAs. We will work out a common interface through discussions with the community, to be used in both YARN-5468 and YARN-4902.

Operator constraints and limits

The cluster operator can impose its own constraints, which can override the ones submitted by the applications. This way, the cluster operator can impose limits to the applications. For example, consider an LRA that imposes the cardinality constraint `max-cardinality(zk, zk, rack, 5)`, whereas the cluster operator has added the constraint `max-cardinality(zk, zk, rack, 2)`. In this case, no more than two `zk` containers will be allowed in a rack.

Internal representation

Although an application can use the three types of constraints shown above (i.e., affinity, anti-affinity, cardinality), we have found that internally just a single type of constraint is sufficient to capture all cases. In particular, we use the following constraint type:

`{container-group, {target-group, c_min, c_max}, node-group},`

which means that each of the containers having an allocation tag `container-group` should be placed within the same `node-group` with no less than `c_min` and no more than `c_max` containers having an allocation tag `target-group`.

For example, constraint `affinity(storm, hbase-rs, rack)`, is represented internally as `{hbase-rs, {hbase-rs, 1, MAX_INT}, rack}`.

All affinity, anti-affinity and cardinality constraints can be expressed using the above single constraint expression. This observation simplifies the code internally. However, we surface the three different constraint types, as they are simpler for the end user.

4. Scheduling an LRA

We have introduced two main new components in YARN: the **Constraint Manager** and the **LRA Interceptor**. The system architecture is depicted in Figure 2.

The Constraint Manager stores the placement constraints for each LRA, the tags of each active container in the system, as well as the node groups (which are updated by the cluster operator, and initially include simply nodes and racks).

The LRA Interceptor uses the interceptor abstraction we are introducing in the ApplicationMasterService (YARN-6355). This taps into the interceptor model we first introduced in the NodeManager (YARN-2884), and can also be useful for YARN-1547.

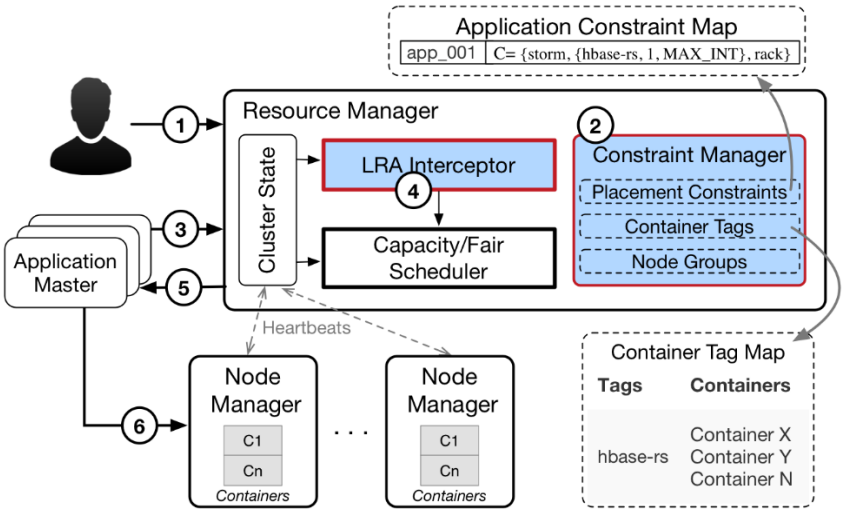

Figure 2 System architecture.

Life-cycle of Storm application

Consider a Storm application, whose life-cycle goes through the following steps (see also Figure 2):
Step 1. The application gets submitted to the cluster. In its ApplicationSubmissionContext, we add the following constraint to impose that it is not placed at the same rack running containers marked with an allocation tag `hbase-rs` (i.e., rack anti-affinity to HBase Region Servers):

```
ApplicattionSubmissionContext {
    ...,
    constraintExpression: "anti-affinity(storm, hbase-rs, rack)"
}
```

Step 2. Once the RM receives the ApplicationSubmissionContext, it translates the constraint to its internal representation, resulting in the following constraint that gets added to the Constraint Manager:

```
{storm, {hbase-rs, 1, MAX INT}, rack}
```

Step 3. The AM for this Storm application gets initialized, and sends the following AllocateRequest to the RM, to start 5 Storm workers, marking them with the `storm` allocation tag:

```
AllocateRequest {
    ResourceRequest { ..., numContainers=5, allocationRequestID=05 },
    AllocationTagsMap { { 05, { storm } } }
}
```

Step 4. Given that the above AllocateRequest has allocation tags attached to it, instead of going directly to the main (Capacity or Fair) Scheduler, it gets handed to the LRA Interceptor. The Interceptor queries the Constraint Manager to check if there are constraints specified for this application, and gets back the constraint that imposes rack anti-affinity to HBase Region Server. Then, again through the Constraint Manager, it finds the nodes in which HBase Region Servers are running, and taking also into account the cluster's available resources, it determines the nodes in which the 5 Storm containers should be placed. Subsequently, it updates the ResourceRequest to request containers on those specific nodes, and sends the AllocateRequest to the main scheduler, who performs the allocation. If the resources at the specific nodes are no longer available, the allocation either fails (in which case the interceptor gets re-invoked), or, in case the user opts for relaxed locality, the container might be placed in another node of the same rack.

Step 5. The RM returns the containers to the AM, who matches them based on the AllocationRequestID and the allocation tags.

Step 6. Once informed of the container allocation, the AM dispatches the tasks to the specific nodes to start their execution.