June
2015

Extend YARN to Support Distributed Scheduling

KONSTANTINOS KARANASOS, SRIRAM RAO, CARLO CURINO, CHRIS DOUGLAS

Table of Contents

Motivation 2
Design 2
Overview 2
Resource Request 3
Container Allocation 4
Task Execution 4
Implementation Details 4
Introducing Container Types [YARN-2882] 4
Proxying all AM-RM Communications [YARN-2884] 5
LocalRM: Distributed Scheduling for QUEUEABLE Containers [YARN-2885] 5
Queuing of Container Requests in the NM [YARN-2883] 5
Estimating Waiting Time in NM Container Queues [YARN-2886] 6
AM Policies for Choosing Types of Containers [YARN-2887] 7
Corrective Mechanisms for Rebalancing NM Container Queues [YARN-2888] 7
Limit in the Number of QUEUEABLE Container Requests per AM [YARN-2889] 7

Motivation

YARN follows a centralized approach for sharing the resources of a cluster among different users and applications. In particular, YARN's Resource Manager (RM) is the central component with which all Application Masters (AMs) negotiate for acquiring cluster resources.

Having the global view of the cluster, YARN's RM can make high-quality placement decisions for heterogeneous applications, taking into account locality constraints and balancing the cluster's load. Moreover, the current design is ideal for enforcing sharing invariants, such as fairness (using Hadoop's Fair Scheduler) or capacity (using the Capacity Scheduler), which is crucial when multiple users are sharing the cluster. However, this centralized architecture puts the RM at the critical path of all allocation decisions, leading to higher allocation latencies, especially as the cluster size and number of submitted applications increases.

At the other end of the design spectrum, distributed resource managers are the leading alternative for offering low allocation latency and, thus, high scheduling throughput. They typically allow multiple schedulers (either one per running job or one per node) to perform independent scheduling decisions and to queue tasks directly at the worker nodes. Despite its advantages, this design relies on a uniform workload, as all schedulers have to run the same scheduling algorithm. Moreover, the local scheduling decisions are often not globally optimal and the queuing at the end nodes may lead to unpredictable job performance. What is more, supporting fairness/capacity guarantees in this context is not trivial.

In this document, we argue that these two solutions are complementary, and propose a blended approach that combines the best of both worlds. Concretely, while preserving YARN's central RM, we add a scheduler to every cluster node that can make low-latency distributed scheduling decisions. We expose this flexibility to the applications, which can now choose the scheduling type (centralized or distributed) for each of their tasks, depending on their needs. Intuitively, opportunistic jobs or jobs with short tasks can benefit from distributed scheduling the most. Our goal is to support diverse application frameworks, provide high cluster throughput with low-latency allocations whenever needed, and enforce sharing invariants (capacity/fairness).

In the remainder of the document, we first give a high-level overview of our design, and then give more details regarding our proposed changes.

Design

In this section, we first provide an overview of the system's architecture, then we give more details regarding the resource request, container allocation and task execution.

Overview

The architecture of the system is depicted in Figure 1. Our system is designed as an extension to YARN's architecture in order to support distributed scheduling. One of the main new components of the system is the LocalRM (standing for Local Resource Manager) that is implemented as a service that runs at every Node Manager (NM). Whenever an AM requests resources, instead of contacting the RM (which is what happens in the current version of YARN), it contacts the LocalRM of the same node that the AM is running. Then, depending on the *type* of the AM's resource request, the LocalRM will either forward the request to the RM for scheduling it in a centralized fashion, or it will

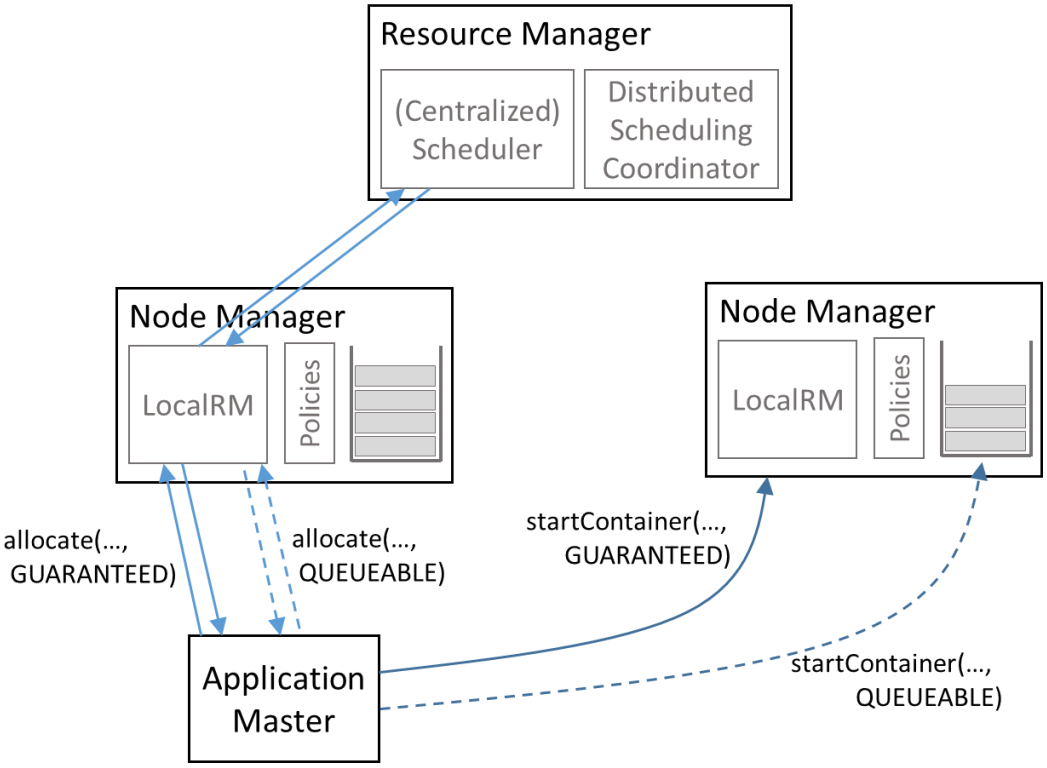

Figure 1. Architecture

act as a distributed scheduler, allowing the AM to immediately send its request for execution to one of the cluster's nodes.

We have also extended the NM by allowing container requests to be queued at each node. This is essential for supporting distributed scheduling: due to the fact that the distributed schedulers may make conflicting allocation decisions, it is not guaranteed that a resource will be immediately available the moment a container request arrives at the NM for execution, thus queuing is required. Moreover, various policies are implemented to determine the behavior of the system, e.g., to decide where the container requests will be allocated during distributed scheduling, to impose quotas on the number of requests a distributed scheduler can send (as a way to throttle down malicious or misbehaving AMs), to rebalance the load of the NM queues, etc.

Finally, a Distributed Scheduling Coordinator component is added to assist the distributed scheduling. This component receives in regular intervals (through the NM-RM heartbeats) the condition of each NM queue and forwards this information to the rest of the nodes in order to help the LocalRMs make educated allocation decisions, thus loosely coordinating the distributed scheduling. The Distributed Scheduling Coordinator is collocated with the RM.

Resource Request

Consider an application running in the cluster (Application Master in Figure 1) that wants to obtain resources. To this end, it petitions the LocalRM of the node the AM was instantiated by using the allocation request API of YARN. This API has been extended so that AMs can also specify the type of the requested container among GUARANTEED or QUEUEABLE. The former are the existing YARN containers that are handed by the RM and offer strict execution guarantees. The latter are the

containers handed by the LocalRMs and are executed opportunistically when there are idle resources in the cluster.

Container Allocation

As already mentioned, when a LocalRM receives a request for GUARANTEED containers, it forwards it to the RM. Then the RM follows the existing procedure in YARN in order to serve this request based on the available cluster resources, the locality constraints of the request and the sharing guarantees of the scheduler that is being used (Fair Scheduler, Capacity Scheduler, etc.). Once there are resources available to meet the requirements of this request, the RM responds to the LocalRM, granting it access to specific cluster resources (containers on specific nodes). The LocalRM forwards this information back to the AM that initiated the request.

On the other hand, when a LocalRM receives a request for QUEUEABLE containers, it locally determines the nodes in which these containers should be enqueued, that is, the nodes that are most likely to be less loaded. This is done by taking advantage of the information given to it by the Distributed Scheduling Coordinator. The LocalRM then informs the AM that initiated the resource request.

In both cases, once the AM gets notified about the containers that were allocated to it, it sends a remote request to start the containers' execution at the remote machines.

Task Execution

When a container request arrives at an NM, its execution is determined by the available resources of the NM and the container type. GUARANTEED containers start their execution immediately, and if needed, the NM will pre-empt/kill already running QUEUEABLE containers to ensure there are sufficient resources for their execution to start. QUEUEABLE containers get placed at the NM's queue. The NM monitors the local resources, and when there are sufficient resources available, it starts the execution of the QUEUEABLE container that is at the head of the queue.

Implementation Details

In this section we focus on various details of the system's implementation. Each subsection corresponds to a sub-JIRA of the umbrella JIRA YARN-2877.

Introducing Container Types [YARN-2882]

We introduce the following two types of containers, each of which is associated with specific allocation/execution semantics:

- **GUARANTEED** containers are allocated by the YARN RM and incur no queuing delay, i.e., their execution is guaranteed to start as soon as they arrive at the NM. Moreover, these containers run to completion bar failures, that is, they are never pre-empted or killed by the infrastructure. These containers correspond to the existing YARN containers.
- **QUEUEABLE** containers are allocated by one of the LocalRMs and enable the AM to "queue" a task for execution on a specific node. No guarantees are provided on the queuing delay, or on whether the container will run to completion or be pre-empted.

When requesting resources, the AM has to specify the container type (the default container type is GUARANTEED). This is done by implementing an AM policy, as will be described below.

Moreover, when a container is sent by the AM for execution to a remote NM, its type is also specified, so that the NM knows how to handle it (whether to execute it immediately or queue it).

Proxying all AM-RM Communications [YARN-2884]

The local Resource Manager proxy (or AMRMProxy for short) is a service running on each node and is implemented as an auxiliary service of the YARN NM. It implements the ApplicationMasterService protocol, just as the YARN RM does. Its role is to act as a proxy between the AMs and the RM, and can be instrumental for enabling various functionality, such as:

- Distributed scheduling (as will be explained in the following subsection);
- RM Federation (see JIRA YARN-2915 for more details);
- Throttling down misbehaving or malicious AMs.

Upon initialization, by rewriting a single configuration parameter, we force the AMs to communicate with the AMRMProxy that is running locally at the same node, instead of the RM. Moreover, to ensure that the AMs will not be allowed to talk directly with the RM, when a new AM gets initialized, we replace its AMRMToken with a token signed by the AMRMProxy.

LocalRM: Distributed Scheduling for QUEUEABLE Containers [YARN-2885]

The LocalRM extends the AMRMProxy and is responsible for the distributed scheduling of the QUEUEABLE containers. In particular, the AMs send their resource requests for allocating containers to the LocalRM.

Requests for GUARANTEED containers are forwarded to the RM. Therefore, in this case the LocalRM acts simply as a proxy between the AM and the RM. It also forwards back to the AM the response of the RM, informing it for allocated GUARANTEED containers.

Requests for QUEUEABLE containers are handled locally and are not forwarded to the RM. The LocalRM receives in regular intervals information about the *k* least loaded nodes in the cluster through the Distributed Scheduling Coordinator, as described in the Design Section. Then it picks among these *k* nodes in a pseudo-random way to allocate QUEUEABLE containers to the AMs that requested them.

Note that in YARN, for security reasons, whenever the RM grants permission to an AM to start a container at a specific node, it also provides the AM with a signed token (NMToken). Without this token, the AM cannot start the container at that node. This prevents unauthorized AMs from arbitrarily submitting containers for execution to the cluster nodes. Respecting this token-based security model, when the LocalRM allocates a container to an AM for execution at a node, it also provides the AM with a locally signed token for that container (using a QueuableTokenProvider). This way the container can pass the authentication test at the remote NM and successfully start its execution.

Queuing of Container Requests in the NM [YARN-2883]

The RM, as well as the LocalRM of each of the nodes, independently allocate containers on a single shared pool of machines. This in turn means that conflicting allocations can be made by the schedulers, potentially causing resource contention. To this end, we extended the NM to add the capability for queuing containers for execution.

Notice that queuing was not needed in the existing version of YARN, since all resources are allocated by the RM, and whenever a container is sent for execution to a node, this node is guaranteed to

have resources available for immediate execution (otherwise, it will not have been allocated there in the first place).

In case of resource contention, we solve the possible conflicts between two containers, taking into account their type, as follows:

- **GUARANTEED - GUARANTEED** By design the RM prevents this type of conflicts by linearizing allocations. This is done by allocating a GUARANTEED container only when it is certain that the target node has sufficient resources, which is what already happens in the existing YARN design.
- **GUARANTEED - QUEUEABLE** This occurs when the RM and one or more of the LocalRMs allocate containers on the same node, causing the node's capacity to be exceeded. Following container semantics described above, any cross-type conflict is resolved in favor of GUARANTEED containers. In the presence of contention, (potentially all) running QUEUEABLE containers are terminated to make room for any newly arrived GUARANTEED. If GUARANTEED containers are consuming all the node resources, the start of QUEUEABLE ones is delayed until resources become available.
- **QUEUEABLE - QUEUEABLE** This occurs when multiple LocalRMs allocate containers on the same target node in excess of available resources. In this case, this node's NM enqueues the requests and thereby prevents conflicts.

The LocalRM periodically checks the condition of the queue and the available resources in the node. Whenever there are queued containers and available resources, the container that is waiting at the head of the queue will be executed.

When a QUEUEABLE container is killed, there is potentially wasted computation. To avoid this, we support promoting a running QUEUEABLE container to a GUARANTEED one. For instance, an AM can automatically promote a resource request in case it has been killed multiple times due to resource contention.

Estimating Waiting Time in NM Container Queues [YARN-2886]

Estimating the waiting time of a container in the NM queue plays an important role in choosing the least loaded nodes to be used during the distributed scheduling. To this end, each NM periodically estimates its waiting queue time and piggybacks this estimate to the NM-RM heartbeat. At the RM, the Distributed Scheduling Coordinator uses this information to pick the *k* least loaded nodes. Using the subsequent heartbeat response from the RM to the NMs, this list of least loaded nodes is pushed to all the LocalRMs.

We now describe how each NM performs the queue waiting time estimation. First, the head-of-line blocking at a node is estimated based on: (1) the cumulative execution times for QUEUEABLE containers that are currently enqueued (denoted Tq), (2) the remaining estimated execution time for running containers (denoted T_r). To enable this estimation, individual AMs provide task run-time estimates when submitting containers for execution. Second, we use the elapsed time since a QUEUEABLE container was last executed successfully on a node, denoted T_l, as a broad indicator of resource availability for QUEUEABLE containers on that node. The NM determines the ranking order R of a node using the following formula:

$$R = T_q + T_r + T_l$$

AM Policies for Choosing Types of Containers [YARN-2887]

As already explained, we expose the API for applications to request both GUARANTEED and QUEUEABLE containers. To take advantage of this flexibility, each AM should implement an AM policy that determines the desired type of container for each task. These policies allow users to tune their scheduling needs, going all the way from fully centralized scheduling to fully distributed (and any combination in between). As a first AM policy implementation, we introduce the following flexible policy.

hybrid-GQ is a policy that takes two parameters: a task duration threshold t_d, and a percentage of QUEUEABLE containers p_q. QUEUEABLE containers are requested for tasks with expected duration smaller than t_d, in p_q percent of the cases. All remaining tasks use GUARANTEED containers. In busy clusters, jobs' resource starvation is avoided by setting p_q to values below 100%. Note that fully centralized scheduling corresponds to setting $t_d = 0$, and fully distributed scheduling corresponds to setting $t_d = \infty$ and $p_q = 100\%$.

Corrective Mechanisms for Rebalancing NM Container Queues [YARN-2888]

Occasionally poor placement choices for QUEUEABLE containers may be made by the LocalRMs, either due to the nature of distributed scheduling (multiple LocalRMs may place containers at the same NM queue) or due to stale queue time estimates.

To deal with this load imbalance that may be observed between the NM queues, we perform load shedding. This has the effect of dynamically re-balancing the queues across machines. We do so in a lightweight manner using the Distributed Scheduling Coordinator. In particular, while aggregating the queuing time estimates published by each NM, the Coordinator constructs a distribution to find a targeted maximal value. It then disseminates this value to the various NMs through the heartbeat mechanism. Subsequently, using this information, the NM on a node whose queuing time estimate is above the threshold, selectively discards QUEUEABLE containers to meet this maximal value. This forces the associated individual AMs to requeue those containers elsewhere. Observe that these policies rely on the task execution estimates provided by the users. Interestingly, even in case of inaccurate estimates, re-balancing policies will restore the load balance in the system.

Limit in the Number of QUEUEABLE Container Requests per AM [YARN-2889]

Given that all resource requests from the AMs go through the LocalRM, the latter can keep track of how many requests each AM sends. Moreover, the responses from the AM-RM heartbeat that contain information about the running and completed containers per application, also pass through the LocalRM. Combining this information, the LocalRM can enforce quotas on the number of QUEUEABLE containers that each AM can use. These quotas can be either absolute limits (e.g., an AM cannot have more than 100 QUEUEABLE containers at the same time) or relative ones (e.g., an AM cannot have more QUEUEABLE containers than double the number of the GUARANTEED ones).