YARN-1011. Schedule containers based on utilization of currently allocated containers.

Karthik Kambatla

January 13, 2016

1 Motivation

Yarn allocates resources based on the resource availability across nodes in the cluster. Today, a node's resource availability is calculated as its "capacity" (determined by *yarn.nodemanager.resource.**) less any resources allocated to containers on the node. A container's allocation is the upper limit of resources the container could use, and is based on the application's request.

In practice, applications are typically conservative with resource requests leading to (potentially severe) cluster under-utilization. The estimates are conservative because it is hard to predict the exact amount of resources required for an application (job): (1) A job could have multiple tasks; each task's resource needs depend on the input it processes. (2) A job's/task's input (size, skew etc.) could change from run to run, and production jobs need to provision for peak load. (3) A task's usage varies over time.

2 Proposal

Like traditional operating systems, we could address this under-utilization by having the scheduler consider the actual utilizaiton on each of the nodes. One could over-subscribe the nodes since most containers under-utilize the allocated resources. However, once an over-subscribed node is fully utilized, a container might need more resources upto its allocation adversely affecting other containers or its own execution.

In YARN-1011, we seek to implement basic over-subscription of nodes with proactive, graceful handling of variations in resource usage. At a high-level, we propose:

1. The notion of OPPORTUNISTIC containers that can soak up the un-utilized resources on a fully-allocated node. (YARN-2882)
2. Nodes to monitor and report container- and container-aggregate- utilization to the scheduler. (YARN-3534, YARN-1012, YARN-3980)
3. Provide a way to turn on over-allocation on a per-node basis. When turned on,
   (a) Launch opportunistic containers at a lower priority to address cases where monitor-and-preempt doesn't kick in soon enough. More details in Section **?**.
   (b) When the utilization goes over a configurable threshold, the NM should preempt enough OPPORTUNISTIC containers to bring the utilization under the threshold.
   (c) When a GUARANTEED container finishes, attempt/request to "promote" one or more of the OPPORTUNISTIC containers.
4. For heartbeats from nodes with over-subscription turned on, the scheduler allocates OPPORTUNISTIC containers based on a configured policy. More details in Section **?**

3 Details

Think of this section as a constant work-in-progress. Will update this based on the discussions on the JIRA.

3.1 Scheduling

For better resource utilization, we might want to maximize the number of OPPORTUNISTIC containers. However, running too many OPPORTUNISTIC containers leads to a higher likelihood of maxing out the resources on the node and having to preempt one/more of them; preempted containers are wasted work - the corresponding resources donot contribute to meaningful utilization. Furthermore, these OPPORTUNISTIC containers could potentially interfere with GUARANTEED containers on "resources" we might not be able to enforce isolation on — say, page faults.

Ideally, we want to run just enough OPPORTUNISTIC containers that maximizes "meaningful" utilization improvements. We will have to rely on heuristics since it is hard to predict the exact resource usage of containers. The choice of heuristic would depend heavily on the cluster and workload in question. We defer implementing these heuristics for later versions, but introduce two coarse-grained knobs these heuristics may want to tune:

- *yarn.nodemanager.overallocation.allocation-threshold*: Node utilization threshold **upto** which the scheduler allocates OPPORTUNISTIC containers.
- *yarn.nodemanager.overallocation.preemption-threshold*: Node utilization threshold **beyond** which the node preempts OPPORTUNISTIC containers in LIFO order.

The gap between the two thresholds (preemption-threshold $>$ allocation-threshold) serves as a buffer to accommodate any increase in container utilization; the wider it is, the less likely it is that we need to preempt any containers. The preemption-threshold is set to 1 by default — start preempting containers only when the node is fully utilized. Setting it to a slightly lower value gives the NM enough time to actively preempt the "right" container instead of letting the OS handle it.

3.1.1 Promotion

When resources become available in the cluster (for some node, total allocation is less than advertised capacity), an OPPORTUNISTIC container may be "promoted" to a GUARANTEED container so it is isolated from other OPPORTUNISTIC containers. While promotion within a node is desirable, the benefits of promoting a container from one node to another are debatable and depends on the current progress of that task.

In Phase-1, we choose to avoid cross-node promotions altogether for two reasons: (1) determining the progress and then whether to promote or not is quite involved, (2) nodes preempt containers when there is resource contention; if an OPPORTUNISTIC container is not preempted, that node has no resource contention issues and preempting the container to promote to another node is likely not going to change much.

3.1.2 Phase-1 scheduling policy

1. On a node update, after processing completed containers, the scheduler attempts to promote enough currently running OPPORTUNISTIC containers to meet the un-allocated resources vacated by completed containers.
2. On a node update, the scheduler updates the node's availableResources based on its utilization and over_allocation_threshold, and update the total cluster capacity accordingly. This way, the scheduler can continue allocating containers the same way it does today.
3. On *SchedulerNode#allocateContainer*, mark the container OPPORTUNISTIC if allocating this container would take the allocation over the advertised capacity. And, keep track of opportunistic containers in a list separate from launchedContainers.
4. If a node kills an OPPORTUNISTIC container due to contention, the node reports this on the update so the RM can update its records.

3.2 Run OPPORTUNISTIC containers at a lower priority

Even though we monitor the node's usage and preempt OPPORTUNISTIC containers reactively, GUARANTEED containers might spike their usage in between two monitoring intervals. In these cases, the OPPORTUNISTIC containers should not affect the execution of GUARANTEED containers. In other words, there should be no impact of OPPORTUNISTIC containers on GUARANTEED containers at any point in time.

In YARN-1011, we restrict ourselves to Linux. We propose to achieve this prioritization through cgroups and other Linux-specific mechanisms.

3.2.1 CPU

To ensure OPPORTUNISTIC containers do not interfere with the execution of GUARANTEED containers, we could set *cpu.shares* for OPPORTUNISTIC containers to the lowest value (2). The GUARANTEED containers will continue to use *1024 * allocated-vcores*.

Note that over-subscription will be disabled when strict CPU management (*y.nm.lce.cgroups.strict-resource-usage*) is turned on.

3.2.2 Disk and Network

In theory, disk and network should be handled the same way CPU is. Need to flesh the details out, based on the latest state of their isolation in YARN.

3.2.3 Memory

Memory is different from other resources: (1) memory usage of processes is not bursty, (2) memory is not a malleable resource. If the aggregate memory usage of the node goes over the configured threshold, it is important to just kill enough OPPORTUNISTIC containers.

We propose to do the following (we use memory cgroups implemented in YARN-1856):

1. Configure the OOM Killer priorities so the OPPORTUNISTIC containers are killed first when we run out of memory.
2. Cgroups soft limit: When a node is under memory pressure, the system tries to limit the memory usage of processes to the soft limit. We could set the soft limits to '0' and allocated-value for OPPORTUNISTIC and GUARANTEED containers respectively.
3. Cgroups swap limit: We could set the swappiness of OPPORTUNISTIC containers to ¿100 so the memory used by these processes can be aggressively swapped.
4. Cgroups OOM control: When enabled, a process that goes over its usage is killed. When disabled, the process is paused until more memory is available. We could enable this for OPPORTUNISTIC containers and disable for GUARANTEED containers.
5. In addition to these, we could get the aggregate usage across containers from the YARN cgroup's *memory.usage_in_bytes*.

4 Out of scope (a.k.a Future work)

In the interesting of putting together a usable version at the earliest, this JIRA aims to implement the simplest version of over-subscription. The following items are out of scope and reserved for future work (in no order of priority):

1. Support for non-Linux operating systems.
2. Sophisticated policies and heuristics to (1) pick applications more conducive to be run using OPPORTUNISTIC containers, (2) dynamic thresholds for the extent of over-subscription.