EXPLANANTIONS OF GLOBAL SCHEDULING (YARN-5139) IMPLEMENTATION

Wangda Tan

Overview

This doc is to explain implementation details of YARN-5139 since size and complexity of the patch is a little hard to be reviewed.

So I still suggest read the Design doc of YARN-5139 before reading this implementation explaination doc.

Before this patch, scheduler in YARN lives in a single thread world: all write ops are protected by the top level scheduler synchronized lock.

What proposed in the patch is to split the **scheduler allocation** process into two parts:

1. Read-only operation, include but not limited to:
    - Check for queue/user/application limits
    - Select most deserving queue / application / request
    - Match best node(s) for given resource request
    - Check locality delay, node label, etc.
2. Write-only operation, include but not limited to:
    - Update resource usage and related metrics/states for queue/user/app
    - Update pending resource request table for applications

With this, what we will do is:

1. Read-only operations will be parallelly processed

    Multiple threads can look up the scheduler structure concurrently, they can figure out, for next most deserving resource request, which is the best node to allocate. We call this "resource allocation proposal" (or proposal for short).

2. Write-only operation will be serially processed like before

Scheduler will process generated resoruce allocation proposal one by one, it can either accept and update internal states, or it will reject and not touch internal states.

Back To The Implementation

Java Object Definitions

There're several new objects will be used:

```
/*
 * This is the resource allocation proposal, each proposal includes containers to be:
 * allocated / reserved / released
 */
class  ResourceAllocationCommitRequest {
    List<AllocationProposal> containersToAllocate;
    List<ReservationProposal> containersToReserve;
    List<ContainerId> containersToRelease
}

/*
 * Usable nodes from scheduler (for example, which has available resource > 0)
 */
class  PlacementSet {
    Map<NodeId, SchedulerNode> nodeSet;
}
```

Top Level Scheduler Changes

In top level scheduler, changes look like

```
/*
 * Handle a ResourceAllocationCommitRequest, can either accept it or reject it
 */
interface  ResourceAllocationCommitter {
    void  handle(ResourceAllocationCommitRequest request);
```

```
}

/*
 * Scheduler will implement the interface
 */
class Scheduler implements ResourceAllocationCommitter {
    ResourceAllocationCommitRequest getAllocationProposal(PlacementSet clusterPlacementS
      readLock {
        // Following operations are in read-lock
        return rootQueue.getAllocationProposal(clusterPlacementSet);
      }
    }

    void tryCommit(ResourceAllocationCommitRequest proposal) {
      writeLock {
        // Following operations are in write-lock

        // Get application for a given proposal
        SchedulerApplicationAttempt app = get_application(proposal);

        // Check if the proposal will be accepted or not:
        boolean accepted = app.accept(proposal);

        if (accepted) {
          // If proposal is accepted, apply the proposal (update states)
          // The reason why we need two separate accept / apply method is:
          // We need first check if proposal can be accepted before update
          // internal data. Otherwise we need revert changes if proposal is
          // rejected by upper level.
          app.apply(proposal);
        } else {
          // Otherwise, discard the proposal
        }
      }
    }
```

```
    // We can have multiple such thread running at the same time
    Thread allocationThread = new Thread() {
      void run() {
        while (true) {
          ResourceAllocationCommitRequest proposal =
            // Pass down cluster-placement-set, which is essentially a set of all
            // the available nodes in the cluster
            scheduler.getAllocationProposal(get_available_placement_set());
          scheduler.tryCommit(proposal);
        }
      }
    }
}
```

Application & Queue Level Changes For GetAllocationProposal

They will be very similar to what we have today for container allocation, but only read-only operations. From top to bottom:

ParentQueue:

```
class  ParentQueue {
  ResourceAllocationCommitRequest getAllocationProposal(PlacementSet clusterPlacementSet
      readLock {
          // All allocations are under read-lock

           if  (!exceed_queeu_max_limit()) {
              return  NOTHING_ALLOCATED;
           }

           for  (queue  in  sort(childQueues)) {
              ResourceAllocationCommitRequest proposal = queue.getAllocationProposal(cluste
              if (proposal != NOTHING_ALLOCATED) {
                  return proposal;
              }
           }
      }
    }
}
```

LeafQueue:

```
class LeafQueue {
  ResourceAllocationCommitRequest getAllocationProposal(PlacementSet clusterPlacementSet
      readLock {
          // All allocations are under read-lock

          if (!exceed_queeu_max_limit()) {
            return NOTHING_ALLOCATED;
          }

          for (application in sort(applications)) {
              if (!exceed_user_limit(application.get_user())) {
                continue;
              }

            ResourceAllocationCommitRequest proposal = application.getAllocationProposal
            if (proposal != NOTHING_ALLOCATED) {
               return proposal;
            }
          }
      }
    }
}
```

Application:

```
class SchedulerApplicationAttempt {
  ResourceAllocationCommitRequest getAllocationProposal(PlacementSet clusterPlacementSet
      readLock {
          // All allocations are under read-lock
           for (request in sort(resource_requests)) {

              // Given request and placement set, find best node to allocate

              // Filter clusterPlacementSet by given resource request, for example:
              // - Hard locality
              // - Anti-affinity / Affinity
              PlacementSet filteredPlacementSet = filter(clusterPlacementSet, request);

              // Sort filteredPlacement according to resouce-request
              for (node in sort(filteredPlacementSet, request)) {
                  if (node.has_enough_available_resource()) {
                      // If node has enough available resource to allocate this request
                      // Return a proposal for allocate this container
                  } else {
                      // If node doesn't have enough available resource
                      // Return a proposal for reserve the container
                  }

                  // Also, what could happen:
                  // - Container released, for example, unnecessary reserved container
                  // - Cannot find node, return NOTHING_ALLOCATED
              }
           }
      }
  }
}
```

Application & Queue Level Changes For Aceept / Apply Reosurce Allocation Proposal

From bottom to top:

Application:

```
class SchedulerApplicationAttempt {
  boolean accept(ResourceAllocationCommitRequest proposal) {
      readLock {
          // Check following:
          // - Does node still have available resource to allocate container
          // - Is node reserved by other application
          // - Is the application still need the resource

          if (canAccept(proposal) && parent.accept(proposal)) {
              return true;
          } else {
              return false;
          }
      }
  }

  void apply(ResourceAllocationCommitRequest proposal) {
      writeLock {
          // Update following:
          // - Deduct pending resource
          // - Update live containers map
          // - Update metrics, etc.
      }

      parent.apply(proposal);
  }
}
```

Leaf/ParentQueue

```
class LeafQueue/ParentQueue {
  boolean accept(ResourceAllocationCommitRequest proposal) {
      readLock {
          // Check following:
          // - queue-limit / user-limit

          if (canAccept(proposal) && parent.accept(proposal)) {
              return true;
          } else {
              return false;
          }
      }
    }

  void apply(ResourceAllocationCommitRequest proposal) {
      writeLock {
          // Update following:
          // - Update resource usages for queue / user
      }

      parent.apply(proposal);
    }
}
```

Plan

To make a step-by-step plan, we need to split tasks, here's the proposal for task split

1. Prerequisite: Optimize synchronized lock in scheduler / queue / application to read/write lock
2. Add interface of ResourceAllocationCommitter
3. Implement ResourceAllocationCommitter and related logics
4. Add PlacementSet and score nodes implementation.