ENHANCE ALLOCATE PROTOCOL TO IDENTIFY REQUESTS EXPLICITLY

Subru Krishnan, Arun Suresh

MOTIVATION

Currently when *AM* sends *ResourceRequests* to the *RM*, with relax locality set to "true", the requests are expanded to into host/rack/any format as illustrated below:

AM request:

Req1 - [1 * Node1/Rack1, 1 * Node2/Rack1, 1 * Node3/Rack2]

Expanded requests:

Node Local: 1 * (Node1), 1 *(Node2), 1* (Node3)

Rack Local: 2 * (Rack1), 1 * (Rack2)

Any : 3 * (ANY)

AM request:

Req2 - [1 * Node1/Rack1, 1 * Node4/Rack1, 1 * Node5/Rack2]

Expanded requests:

Node Local: ***2*** * (Node1), 1 * (Node4), 1 * (Node5)

Rack Local: ***3*** * (Rack1), ***3*** * (Rack2)

Any : ***6*** * (ANY)

As seen above the expansion makes it impossible to track back to the original request (No way to know if an allocation response is to be matched against Req1 or Req2). The correlation becomes increasingly difficult / impossible if Req1 and Req2 are for 1260MB and 1516MB memory respectively, and if the minimum allocation is set to 2048MB (same with integral roundup if minimum allocation is 1024MB), it'll get back 2 containers of 2048MB. So framework AMs like Slider/REEF use priority as a hack to map their requests. Moreover, the expansion and the corresponding reconciliation on Container allocation is managed by the *AMRMClient* which makes it cumbersome for non-Java clients as they all have to replicate the non-trivial logic.

The raw ResourceRequest is required for the following features under development:

- Allow ResourceRequests of different sizes for same priority and location (YARN-314)
- Enable framework AMs like REEF to map allocated containers to original request
- Affinity / anti-affinity and gang scheduling decisions for long running services (YARN-1042)
- Tracking container allocation latencies (YARN-4485)
- FairScheduler preemption decisions (YARN-2154)

- In Federation (YARN-2915) to decide which RM to route the request to.
- For making distributed scheduling decisions (YARN-2877)

PROPOSED APPROACH

At high level, we propose to add an ID field to ResourceRequest - this can be a sequence number for each application. The application should simply generate a unique identifier within the application - if not the client-libraries can do so if desired by the application. This ID is a unique identifier for different ResourceRequests from the *same application* - essentially IDs can conflict across applications. The allocated Container(s) received as part of the response will have the ID corresponding to the original ResourceRequest for which the RM made the allocation. The scheduler may return multiple responses corresponding to the same Request-ID - as and when scheduler returns containers. On AM restarts, a subsequent attempt could choose to resume from appropriate sequence number. Applications can continue to completely ignore the returned Allocation-ID in the response and use the allocated containers for any of their outstanding requests.

IMPLEMENTATION DETAILS

1. Any AM which wants to use the delta protocol will tag it's ResourceRequest with an ID. For e.g.: If it initially wants 3 containers, it'll send a ResourceRequest with ID '1' for 3 containers

2. If the ResourceRequest has a ID, the AMRMClient will simply forward the raw request to the RM without expanding

3. The resource-request data structure in RM's AppSchedulingInfo will be updated to Map<Priority, Map<String, Map<long, ResourceRequest>>>.

4. If the ResourceRequest has a ID and has relax locality set to true, then the RM will expand the ResourceRequest and add it to the host/rack/any with the ID as key. If locality is not relaxed, RM will add an entry corresponding to the ID to only the node (or rack).

5. The RM will attempt to allocate containers in decreasing sequence number order, i.e. FIFO mode. The containers are matched from the largest ID to the smallest ID until the outstanding requests are met. On allocation of the last container of a ResourceRequest , the RM delete the entry for the corresponding ID altogether from AppSchedulingInfo. For e.g.: If an AM asks for 3, and 2 containers of same capability in subsequent allocate calls and the RM has allocated it 3 containers, the AM will have 2 outstanding containers with ID '1'.

   Note: The RM currently does accounting only on the number of outstanding ANY requests, while with the new approach we can update the rack & host requests too as we have mapped them to the same ID. This though very useful can be less performant in the face *of delay scheduling* as the node (or rack if it happens to the last container for the rack) that was decremented loses further scheduling opportunities.

6. If the AM then wants 4 more containers, it'll send a ResourceRequest with ID '2' for 4 containers.

7. The RM will add a new entry to its resource-request data structure for ID '2'.

8. In case an AM wants to update (increase or decrease) one of its previous asks, it sends a ResourceRequest with the ID it originally specified and RM will consequently replace the corresponding entry its resource-request data structure with the new ask. For e.g.: If the AM wants 2 more containers of the same capability as ID '1', it will send another ResourceRequest with ID '1' for 5 containers. The AM can also choose to send a separate ResourceRequest with ID '3' for 3 containers if it doesn't need to logically group the containers.

    - Similar to the current API, update of only selected fields against a previously existing Request-ID will only update the object (as opposed to replacing it). For e.g.: say a ResourceRequest first gets created with ID '7' and with *"host: *"*. A future ResourceRequest with the same ID '7' but with contents *"rack05: 10"* will only append the rack information to the existing object. This is how one can replace parts of an object and is similar to how the existing per-record-deltas based protocol works.

    - Similarly, if one wishes to replace an entire ResourceRequest corresponding to a specific Request-ID, they can simply cancel the corresponding ResourceRequest and submit a new one afresh.

    Note: This works fine for increase asks but we can experience a timing issue for decrease ask where containers are allocated by the RM just when the AM is requesting a decrease. In fact, this scenario is valid presently and we will rely on the AM returning the extra containers as is done today.

NETWORK/MEMORY OVERHEAD

One of the main considerations for the original implementation of the allocate protocol was network bandwidth, so it was designed be bounded by number of nodes. In our proposal, we could potentially have requests for each container at worst case. This should be safe considering modern 10G networks.

We are adding a map of ID to RM's resource-request data structure which will increase the memory footprint in the worst case as described above. Our argument remains the same – RAM size has increased considerably due to downward cost spiral and so should be manageable.

We are working on preventing DoS attacks on RM (YARN-1547), we propose to additional limit the total number of IDs which an application can use to limit the overhead. Additionally, in practical cases the AMs make a finite number of requests typically based on order of container types in which case the approach will be at par with the existing one.

BACKWARD COMPATIBILITY

The changes we propose are wire compatible as we are only adding an ID field to ResourceRequest and Container.

For logical compatibility:

- If the ResourceRequest has no ID, the AMRMClient will expand and forward the request to the RM exactly as is done currently
- The RM on receiving a ResourceRequest without an ID will understand it is a legacy expanded request and will add them unchanged with a default ID, say -1.

This will ensure that the requests are cumulative for any changed entries and the matching will also be exactly the same as the ID map in RM's resource-request data structure will always have only one entry (corresponding to default ID).