Changing Resources of an Allocated Container in Hadoop YARN

1. Overview

The affected components of this project include:

- AMRMClient
- NMClient
- Resource Manager
- Node Manager

The meaning of resource increase and resource decrease depends on what resource calculator is configured in YARN. Default resource calculator only compares memory, whereas dominant resource calculator uses dominant resource algorithm to compare multi-dimensional resources.

The high level flow of increasing and decreasing resource of an existing container is illustrated in Figure 1 and Figure 2 (positive case only). In essence, the increase of container resource is analogous to allocating a new container with host constraint, and decrease of container resource is analogous to killing a container and take away all its resource.

1.1 Increase Container Size

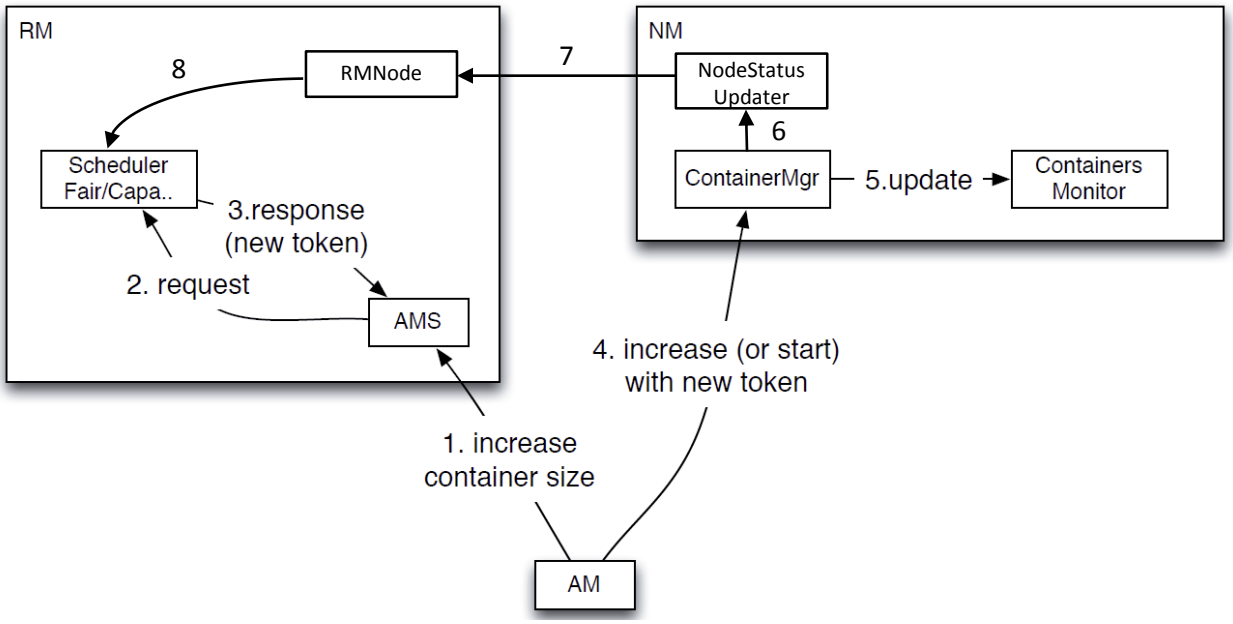

Figure 1 Increasing container resource

High-Level Flow

1. AM sends increase resource request to RM, specifying the container ID
2. RM increases resource in the scheduler following the logic of allocating new containers
3. Once the increase is granted, RM provides a new token to sign the granted increase

4. AM sends increase request to NM, signed with the new token
5. The ContainerManager updates the quota of this container in the ContainerMonitor
6. The ContainerManager sends increase message to NodeStatusUpdater
7. NodeStatusUpdater sends decrease request to RMNode during NM/RM heartbeat
8. RMNode sends increase message to the corresponding scheduler, which will verify the increase request

1.2 Decrease Container Size

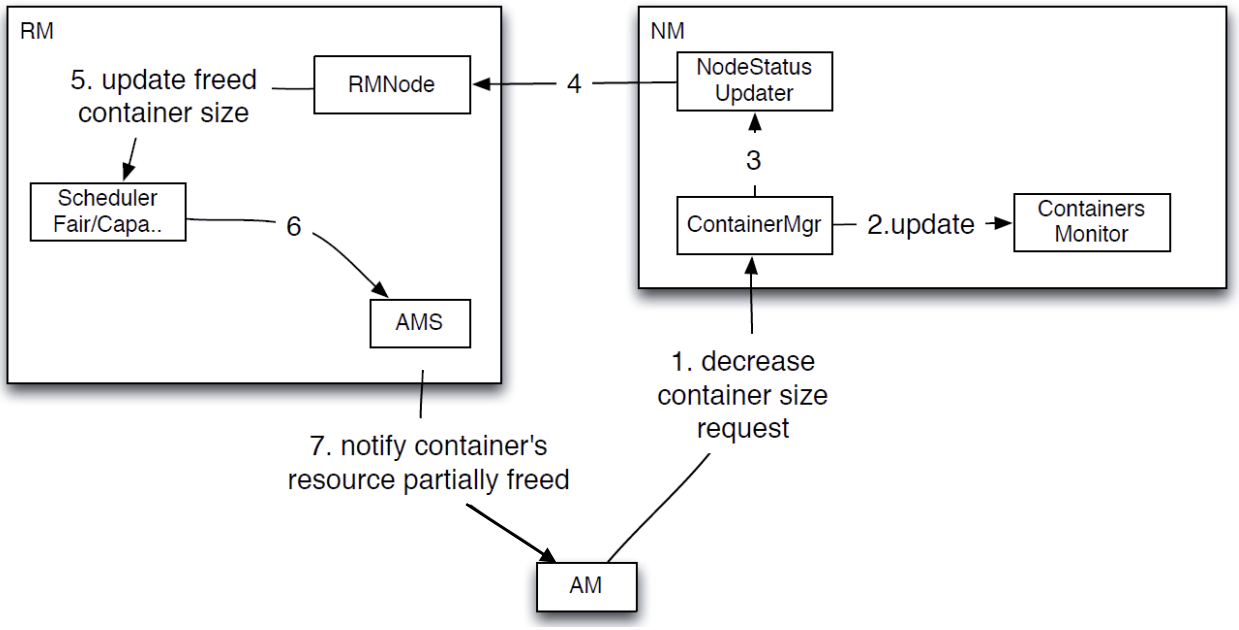

Figure 2 Decreasing container resource

High-Level Flow

1. AM sends decrease resource request to NM
2. The ContainerManager decreases monitoring quota in the ContainerMonitor
3. The ContainerManager sends decrease message to NodeStatusUpdater
4. NodeStatusUpdater sends decrease request to RMNode during NM/RM heartbeat
5. RMNode sends decrease message to the corresponding scheduler
6. RM informs AM about the change and allocates free space somewhere else

To simplify design and avoid race conditions, only containers in **RUNNING** state can have their resource increased or decreased. We do not support increase/decrease containers in ACQUIRED state because:

- Decrease container resource is initiated through the ContainerManager on a NodeManager. If a container is not even started on a node, there is nothing to decrease.
- Increase container resource requires a container token. If a container is in ACQUIRED state, it already possesses a token to launch a new container. In this case, it can be confusing to grant it another token just to increase the container size. Unless this proves to be a really desirable feature, we propose to not allow increasing container resource while it is in ACQUIRED state, as the added effort does not justify the benefits.

2. Design Details

2.1 Protocol

1. New and updated protocols between `ApplicationMaster` and ResourceManager's `ApplicationMasterService`, to send container resource increase requests, and to receive response with containers for which the resource increase requests are granted, or response with containers whose resource have been decreased (via ApplicationMasterProtocolService's `allocate`() rpc service).

yarn_protos.proto

```
message ContainerResourceIncreaseRequestProto {
  optional ContainerIdProto container_id = 1;
  optional ResourceProto capability = 2;
}

message ContainerResourceIncreaseProto {
  optional ContainerIdProto container_id = 1;
  optional ResourceProto capability = 2;
  optional hadoop.common.TokenProto container_token = 3;
}

message ContainerResourceDecreaseProto {
  optional ContainerIdProto container_id = 1;
  optional ResourceProto capability = 2;
}
```

yarn_service_protos.proto

```
message AllocateRequestProto {
  repeated ResourceRequestProto ask = 1;
  repeated ContainerIdProto release = 2;
  optional ResourceBlacklistRequestProto blacklist_request = 3;
  optional int32 response_id = 4;
  optional float progress = 5;
  repeated ContainerResourceIncreaseRequestProto increase_request = 6;
}

message AllocateResponseProto {
  optional AMCommandProto a_m_command = 1;
  optional int32 response_id = 2;
  repeated ContainerProto allocated_containers = 3;
  repeated ContainerStatusProto completed_container_statuses = 4;
  optional ResourceProto limit = 5;
  repeated NodeReportProto updated_nodes = 6;
  optional int32 num_cluster_nodes = 7;
  optional PreemptionMessageProto preempt = 8;
  repeated NMTokenProto nm_tokens = 9;
  repeated ContainerResourceIncreaseProto increased_containers = 10;
  repeated ContainerResourceDecreaseProto decreased_containers = 11;
}
```

The **`ContainerResourceIncreaseProto`** response contains a **`container_token`** which can be used by `ApplicationMaster` in the subsequent calls to NodeManager to initiate the actual container resource increase action.

2. New and updated protocol between `ApplicationMaster` and NodeManager's `ContainerManager` RPC server to initiate the container resource increase action:

yarn_service_protos.proto

```
message ChangeContainersResourceRequestProto {
  repeated hadoop.common.TokenProto increase_containers = 1;
  repeated ContainerResourceDecreaseProto decrease_containers = 2;
}
message ChangeContainersResourceResponseProto {
  repeated ContainerIdProto succeeded_requests = 1;
  repeated ContainerExceptionMapProto failed_requests = 2;
}
```

containermanagement_protos.proto

```
service ContainerManagementProtocolService {
  rpc startContainers(StartContainersRequestProto) returns
(StartContainersResponseProto);
  rpc stopContainers(StopContainersRequestProto) returns
(StopContainersResponseProto);
  rpc getContainerStatuses(GetContainerStatusesRequestProto) returns
(GetContainerStatusesResponseProto);
  rpc changeContainersResource(ChangeContainersResourceRequestProto) returns
(ChangeContainersResourceResponseProto);
}
```

3. Updated `NodeStatusProto` protocols between NodeManager's `NodeStatusUpdater` and ResourceManager's `ResourceTrackerService` to let NM notify RM about container resource changes:

yarn_server_common_protos.proto

```
message ContainerResourceChangeProto {
  optional ContainerResourceIncreaseProto increased_container = 1;
  optional ContainerResourceDecreaseProto decreased_container = 2;
}

message NodeStatusProto {
  optional NodeIdProto node_id = 1;
  optional int32 response_id = 2;
  repeated ContainerStatusProto containersStatuses = 3;
  optional NodeHealthStatusProto nodeHealthStatus = 4;
  repeated ApplicationIdProto keep_alive_applications = 5;
  repeated ContainerResourceChangeProto changed_containers = 6;
}
```

Update the `NodeHeartbeatResponseProto` so that RM can confirm with NM about any resource changes made for a specific container. See section 2.3.3.1 for a detailed explanation on why this field is needed.

yarn_server_common_service_protos.proto

```
message NodeHeartbeatResponseProto {
  optional int32 response_id = 1;
  optional MasterKeyProto container_token_master_key = 2;
  optional MasterKeyProto nm_token_master_key = 3;
  optional NodeActionProto nodeAction = 4;
  repeated ContainerIdProto containers_to_cleanup = 5;
  repeated ApplicationIdProto applications_to_cleanup = 6;
  optional int64 nextHeartBeatInterval = 7;
  optional string diagnostics_message = 8;
  repeated ContainerIdProto containers_to_be_removed_from_nm = 9;
  repeated SystemCredentialsForAppsProto system_credentials_for_apps = 10;
  optional bool areNodeLabelsAcceptedByRM = 11 [default = false];
  repeated ContainerResourceChangeProto containers_to_change = 12;
}
```

2.2 Client

Client APIs need to be updated to facilitate container resource increase/decrease request.

2.2.1 New public APIs in AMRMClient to facilitate the container resource increase request:

```
void addContainerResourceIncreaseRequest(…)
```

In terms of the parameters to this function, there are a few options to wrap the container resource increase request information:

Option 1: Add a new class:

```
public static class ContainerResourceChangeRequest {
    final Resource capability;
    final ContainerId containerId;
    …

}
```

Option 2: Use the existing `ContainerRequest` class, and add a new field `ContainerId`:

```
  public static class ContainerRequest {
    final Resource capability;
    final List<String> nodes;
    final List<String> racks;
    final Priority priority;
    final boolean relaxLocality;
    final String nodeLabelsExpression;
    final ContainerId containerId;
    …
 }
```

Option 3: Provide the container ID and the resource capability directly as parameters to the function.

Since all that is needed for the container resource increase request is a container ID and a desired resource capability, **Option 3** is preferred, as it is much simpler:

AMRMClient.java

```
public abstract void addContainerResourceIncreaseRequest(ContainerId
                          containerId, Resource capability)
```

AMRMClientImpl.java

```
@Override
public synchronized void addContainerResourceIncreaseRequest(ContainerId
                           containerId, Resource capability)
```

AMRMClientAsync.java

```
public abstract void addContainerResourceIncreaseRequest(ContainerId
                          containerId, Resource capability)
```

AMRMClientAsyncImpl.java

```
@Override
public void addContainerResourceIncreaseRequest(ContainerId containerId,
                          Resource capability)
```

This API is only valid when the container of interest is in `RUNNING` state. A Precondition check will be performed on the container state.

Internally, any new container resource increase request will be cached in a map between two heartbeats (i.e., allocate()), and once the requests are sent, the cached map will be emptied.

For recovery purposes, there will also be a map to hold all pending increase requests. A pending increase request is only removed when the request is granted by RM through the heartbeat response. The entire logic is the same as the existing `release` and `pendingRelease` lists.

AMRMClientImpl.java

```
protected final Map<ContainerId, Resource> increase = new HashMap<>();
protected final Map<ContainerId, Resource> pendingIncrease = new HashMap<>();
```

Users who use the `AMRMClientAsync` library in the `ApplicationMaster` need to implement the `onContainersAllocated` callback to check if a container resource increase request has been granted by ResourceManager, and if so, initiate the container resource increase action.

2.2.2 New public APIs in NMClient to facilitate container resource increase and decrease actions:

These APIs are only valid when the container is in `RUNNING` state.

NMClient.java

```
public abstract void increaseContainerResource(Container container);
public abstract void decreaseContainerResource(ContainerId containerId,
                                    NodeId nodeId, Resource capability);
```

NMClientImpl.java

```
@Override
public void increaseContainerResource(Container container);
@Override
public abstract void decreaseContainerResource(ContainerId containerId,
                                    NodeId nodeId, Resource capability);
```

NMClientAsync.java

```
public abstract void increaseContainerResourceAsync(Container container);
public abstract void decreaseContainerResourceAsync(ContainerId containerId,
                                    NodeId nodeId, Resource capability);
```

NMClientAsyncImpl.java

```
@Override
public void increaseContainerResourceAsync(Container container);
@Override
public void decreaseContainerResourceAsync(ContainerId containerId, NodeId
                                    nodeId, Resource capability);
```

2.3 Resource Manager

2.3.1 Accept container resource increase request:

The container resource increase request comes from the `ApplicationMasterProtocol#allocate`. Once the request is received and validated, `ApplicationMasterService` will call `CapacityScheduler`'s allocate method, which in turn will update the outstanding resource increase requests in the corresponding application's `AppSchedulingInfo` class.

A new map `increaseRequests` is introduced in the `AppSchedulingInfo` class to keep track of outstanding resource increase requests:

AppSchedulingInfo.java

```
final Map<NodeId, Map<ContainerId, ContainerResourceIncreaseRequest>>
increaseRequests = new HashMap<>();
```

In between two scheduling cycles, the following rules must apply when updating the `increaseRequests` map:

- New resource increase request only applies to containers in `RUNNING` state.
- New resource increase request to the same container will always overwrite previous resource increase requests if they exist.
- New resource increase request must have resource capacity larger than that of the existing container.
- New resource increase request is subject to the maximum allocation for each container defined in `yarn-default.xml`. Requests higher than the maximum will be capped to the maximum.

2.3.2 Container resource increase and expiration:

The following new events are added to support container resource increase logic:

RMContainerEventType.java

```
public enum RMContainerEventType {
  …
  LAUNCHED,
  FINISHED,
  INCREASE_ACQUIRED,
  INCREASE_CANCELLED,
  INCREASED,

  // Source: ApplicationMasterService->Scheduler
  RELEASED,

  // Source: ContainerAllocationExpirer
  EXPIRE,

  // Source: ContainerResourceIncreaseExpirer
  INCREASE_EXPIRE,
  …
}
```

SchedulerEventType.java

```
public enum SchedulerEventType {
  …
  // Source: ContainerAllocationExpirer
  CONTAINER_EXPIRED,

  // Source: ContainerResourceIncreaseExpirer
  CONTAINER_INCREASE_EXPIRED
}
```

A new `ContainerResourceIncreaseExpirer` is introduced to enforce that the container resource is increased within the expiration interval. The configuration will reuse the value specified with `RM_CONTAINER_ALLOC_EXPIRY_INTERVAL_MS` and `DEFAULT_RM_CONTAINER_ALLOC_EXPIRY_INTERVAL_MS`.

The `ContainerResourceIncreaseExpirer` will track container by its ID and original capacity. In case the granted container resource increase expires, the scheduler needs to release the additional allocated resource from the container.

The container resource increase and expiration logic is illustrated as follows:

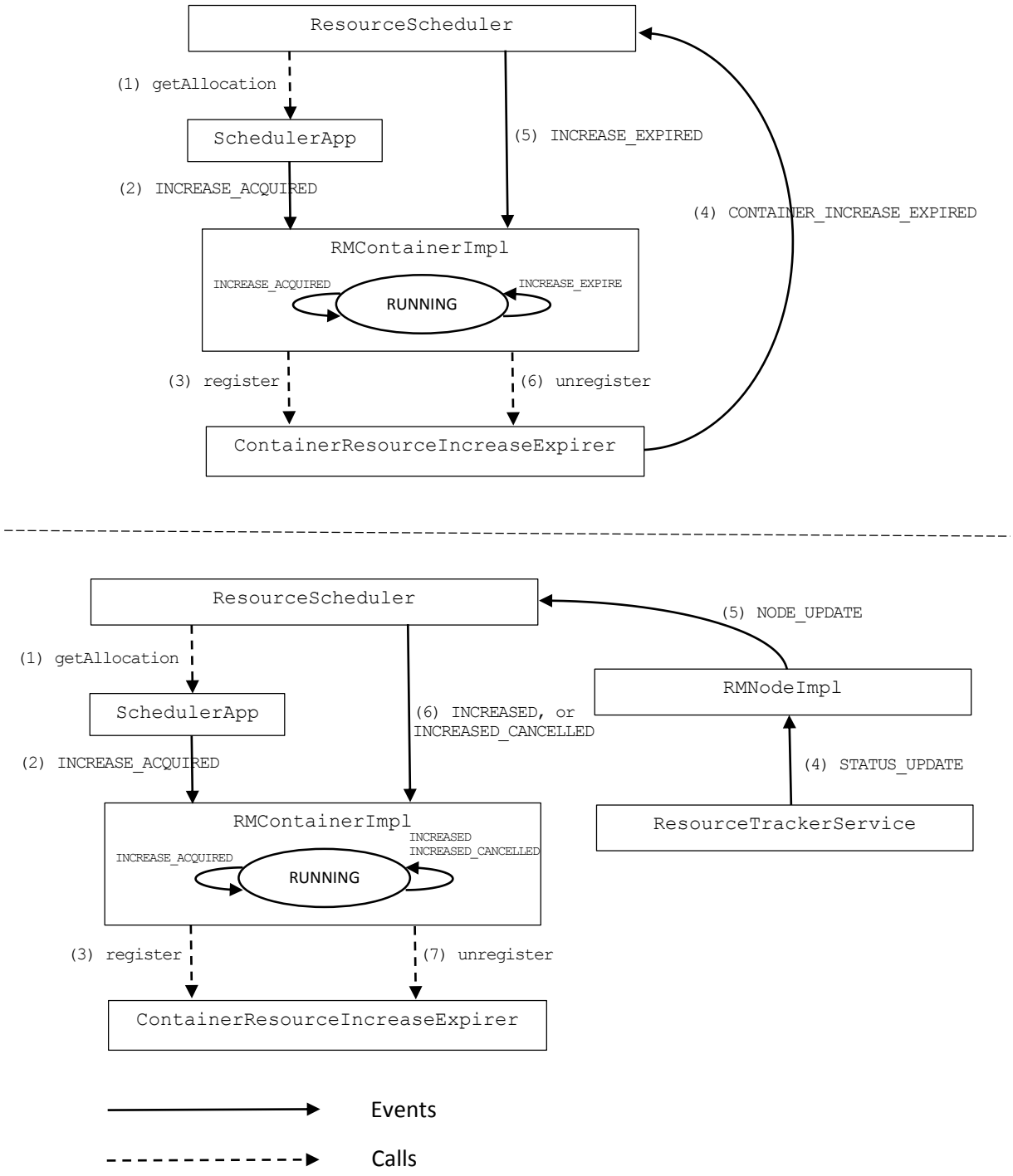

Figure 3. Container Resource Increase and Expiration Logic

2.3.3 Capacity Scheduler

2.3.3.1 NodeUpdate

At each `CapacityScheduler.nodeUpdate`, RM will pull the `newlyChangedContainers` data structures to check any unprocessed container decrease or increase messages coming from NM status update. The NM logic described in section 2.4 makes sure that the container resource change updates from NM heartbeats are processed in the same order of resource change actions initiated from AM to NM.

For a resource decrease message to be valid, the target resource allocation must be smaller than the current resource allocation of the container known by RM.

For a resource increase message to be valid, the target resource allocation must be the same as the current resource allocation of the container known by RM.

Invalid resource decrease messages will be ignored by RM.

Invalid resource increase messages will be ignored by RM, and an `INCREASE_CANCELLED` event will be fired to notify `ContainerResourceIncreaseExpirer` to unregister the increase request.

To process a valid resource decrease message, the scheduler will:

- Decrease container resource in FiCaSchedulerApp/AppSchedulingInfo/FiCaSchedulerNode/LeafQueue/ParentQueue and other-related-metrics. **If, as a result of the decrease, the container resource will go below the minimum allocation for each container specified in `yarn-default.xml`, it will be adjusted to the minimum allocation.** Though this will cause some resource wastes on the node, it will make the implementation of the scheduler simpler.
- Update resource in the container

The decreased container will be set in `AllocateResponse` during the next AM-RM heartbeat.

To process a valid resource increase message, the scheduler will do nothing except to fire an `INCREASED` event to notify `ContainerResourceIncreaseExpirer` to unregister the increase request as illustrated in Figure 3.

After processing all resource change messages for a container in a node update, RM will set the current resource allocation known by RM for this container in the next node heartbeat response, so that NM will (eventually) have the same view of the resource allocation of this container with RM, and monitor/enforce accordingly.

2.3.3.2 Scheduling

Once a node update completes, the scheduling cycle kicks in. The scheduler will first check for reserved application, then do normal allocation. **In both cases, container resource increase request will be considered first before normal container allocation request**.

The scheduling of an outstanding resource increase request to a container will be skipped if there are either:

- Granted resource increase request for that container sitting in RM (i.e., not yet acquired by AM), or
- Granted container resource increase request for that container registered with `ContainerResourceIncreaseExpirer`.

This will guarantee that at any time, there can be one and only one resource increase action being processed for a container.

```
if node.is_reserved():
    // Try to fulfill the reservation
    if reserved-increase-request:
        LeafQueue.assignReservedIncreaseRequest(...)
    elif reserved-normal-container:
        ...
else:
    ParentQueue.assignContainers(...)
    // this will finally call
    // LeafQueue.assignContainers(...)
```

In `LeafQueue.assignContainers`:

```
foreach (application):
    // do increase allocation first
    foreach (increase_request):
        // check if we can allocate it
        // in queue/user limites, etc.
        // return None if not satisfied
        if request-is-fit-in-resource:
            allocate-resource
            update container token
            add to newlyIncreasedContainers
        else:
            reserve in app/node
            return reserved-resource
    // do normal allocation
    ...
```

2.4 Node Manager

1. A new event type will be introduced in `ContainersMonitor` to indicate that the resource usage of a container has been changed:

```
public enum ContainersMonitorEventType {
   START_MONITORING_CONTAINER,
   STOP_MONITORING_CONTAINER,
   CHANGE_MONITORING_CONTAINER
}
```

Correspondingly, a new class `ContainersChangeMonitorEvent` will be introduced to indicate the actual resource that is being changed.

2. All container resource change messages are put in a queue with the same order as the resource change actions, so that when these messages are sent to RM during regular node update heartbeat request, they are processed in the same order.

3. As described in 2.3.3.1, RM will send a confirmation of the resource change result of a container in the node update heartbeat response, so that NM can have a consistent view of the resource allocation of a container with RM, and monitor and enforce accordingly.

4. Resource enforcement

   The following tasks are still under investigation, and will be updated at a later time:

- CGroup memory control of containers in YARN's Linux Container Executor
  The goal is to be able to enforce physical memory consumption through cgroup without having to kill the container when memory usage exceeds the quota.
- Dynamic update of CGroup that a process is run under.

Task list

Component	Task	Priority	JIRA
Common	Protocol Buffer message definition for AM-RM protocol	P1	YARN-1447
	Protocol Buffer message definition for AM-NM protocol	P1	
	Protocol Buffer message definition for NM-RM protocol	P1	
AMRMClient	Protocol Buffer rpc definition change for AM-RM protocol	P1	YARN-1448
	AMRMClient changes to allow sending increase container resource request, and to get container resize response from RM	P1	YARN-1509
NMClient	Protocol Buffer rpc definition change for AM-NM protocol	P1	
	NMClient changes to allow sending container resource increase/decrease action, and to get response from NM	P1	
NodeManager	Add changeContainersResource interface and implementations to ContainerManagementProtocol	P1	YARN-1645
	Update ContainerMonitor to allow changing monitoring size of an allocated container	P1	YARN-1643
	Add newly changed container messages to NodeStatus in NM Process changed container confirmation in the node heartbeat response.	P1	YARN-1644
	Support updating of resource isolation/limit of a container - Dynamic updating of Cgroup - Cgroup memory enforcement	P2	
ResourceManager, Schedulers	Update ApplicationMaster service and YarnScheduler to allow container resource increase requests, and updating of outstanding resource increase requests	P1	YARN-1646 YARN-1648
	Add pullChangedContainers to RMNode to be used by scheduler to get newly decreased and increased containers	P1	YARN-1650
	Add increased/decreased containers to Allocation response	P1	YARN-1647
	Update ResourceTrackerService to support passing of container resource decrease and increase events from NM to RM	P1	YARN-1649

	Add implementation to RMNode and RMContainer to support container resource increase expiration logic	P1	
	Add methods in FiCaSchedulerApp to support container resource changes	P1	YARN-1651
	Add methods in FiCaSchedulerNode to support container resource changes	P1	YARN-1652
	Add APIs in CSQueue to support container resource change request	P1	YARN-1653
	Add implementations to CapacityScheduler to support increase/decrease of container resources	P1	YARN-1654
	Add implementations to FairScheduler to support increase/decrease of container resources	P2	YARN-1655