Changing Resources of an Allocated Container in Hadoop YARN

Authors: Meng Ding, Wangda Tan, Vinod Kumar Vavilapalli

with inputs from Sandy Ryza, Bikas Saha, Alejandro Abdelnur, Siddarth Seth

Last Modified Date: Jun 23, 2015

1. Overview

The affected components of this project include:

- AMRMClient
- NMClient
- Resource Manager
- Node Manager

The high level flow of increasing and decreasing resource of an existing container is illustrated in Figure 1 and Figure 2 (positive case).

1.1 Increase Container Size

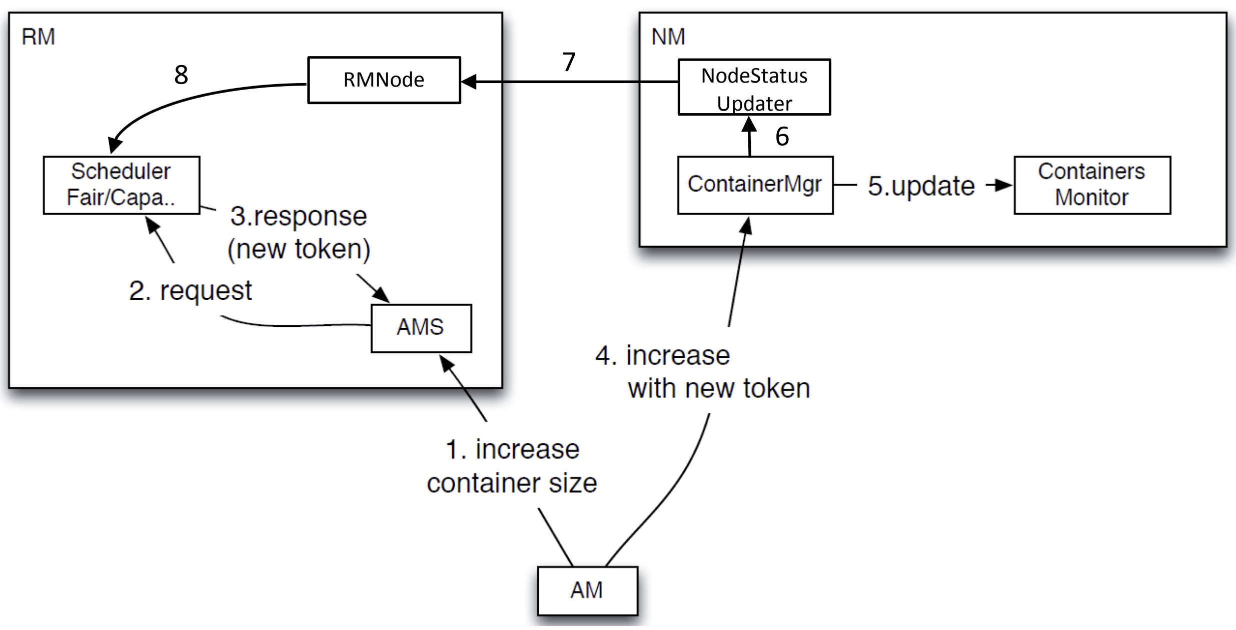


Figure 1: Increasing container resource

High-Level Flow

1. AM sends resource change request to RM, specifying the container ID and the target capability
2. RM increases resource in the scheduler during scheduling cycle following the logic of allocating new containers
3. Once the increase is granted, RM provides a new token to AM to sign the granted increase, and starts the expiration listener
4. AM sends increase request to NM, signed with the new token.
5. ContainerManager informs Container Monitor for resource monitor and enforcement
6. The ContainerManager updates its internal bookkeeping of container resource and metrics info, then sends increase message to NodeStatusUpdater
7. NodeStatusUpdater sends the resource change message to RMNode during NM/RM heartbeat
8. The scheduler unregisters the container from expiration listener, and completes the entire increase cycle

1.2 Decrease Container Size

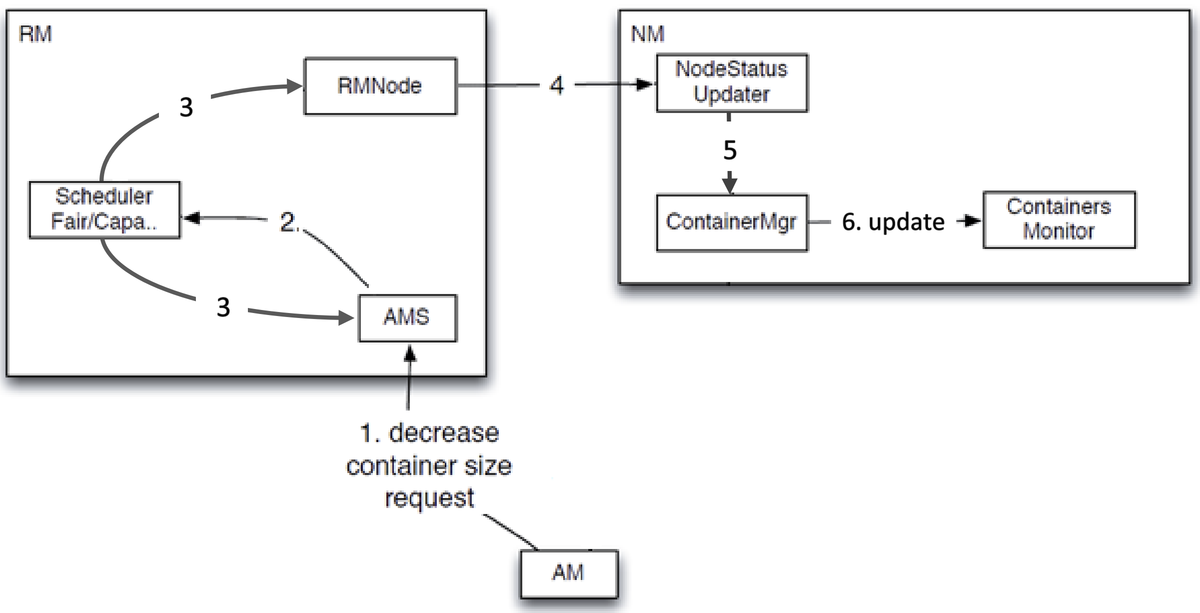


Figure 2: Decreasing container resource

High-Level Flow

1. AM sends resource change request to RM, specifying the container ID and the target capability
2. RM decreases resource in the scheduler during scheduling cycle
3. RM forwards the decreased containers info to RMNode for NM to pull in the next heartbeat. In addition, the decreased container info will be set in the AllocateResponse for AM to pull in the next AM->RM heartbeat
4. NM pulls the decreased containers info from NM->RM heartbeat response
5. NM forwards the decreased container info to ContainerManager, and updates its internal booking keeping of container resource and metrics info
6. ContainerManager informs Container Monitor for resource monitor and enforcement

Both container resource increase and decrease requests will go through Resource Manager. This will avoid race conditions as Resource Manager is the central place to validate, order and synchronize all resource change requests.

To simplify design and avoid race conditions, only containers in **RUNNING** state can have their resource changed. We do not support change a container in ACQUIRED state, because that will involve the invalidation of the existing token of that container and replacing it with a new one. Unless this proves to be a really desirable feature, we propose to not allow it, as the added effort does not justify the benefits.

2. Design Details

2.1 Protocol

1. Add and update protocols between `ApplicationMaster` and ResourceManager's `ApplicationMasterService`, to send container resource change requests, and to receive response with containers for which the resource change requests are approved (via ApplicationMasterProtocolService's `allocate`() rpc service).

```
yarn_protos.proto
message ContainerResourceChangeRequestProto {
  optional ContainerIdProto container_id = 1;
  optional ResourceProto capability = 2;
}
message IncreasedContainerProto {
  optional ContainerIdProto container_id = 1;
  optional ResourceProto capability = 2;
  optional hadoop.common.TokenProto container_token = 3;
}
message DecreasedContainerProto {
  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 ContainerResourceChangeRequestProto increase_requests = 6;
  repeated ContainerResourceChangeRequestProto decrease_requests = 7;
}
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 IncreasedContainerProto increased_containers = 10;
  repeated DecreasedContainerProto decreased_containers = 11;
}
```

The AllocateResponse contains resource change that have been approved by RM:

- The increased_containers in the response contains a container_token which must be used by AM in the subsequent increaseContainersResource rpc call to initiate the actual container resource increase action on Node Manager.
- The decreased_containers in the response contains the ID and capability of the container that has just been downsized.

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

```
yarn_service_protos.proto
message IncreaseContainersResourceRequestProto {
  repeated hadoop.common.TokenProto increase_containers = 1;
}
message IncreaseContainersResourceResponseProto {
  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 increaseContainersResource(IncreaseContainersResourceRequestProto)
returns (IncreaseContainersResourceResponseProto);
}
```

3. Update NodeStatusProto protocols between NodeManager's NodeStatusUpdater and ResourceManager's ResourceTrackerService to let NM notify RM about container increase:

```
Yarn_server_common_protos.proto

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 IncreasedContainerProto increased_containers = 6;
}
```

4. Update the NodeHeartbeatResponseProto so that RM can notify with NM about any resource decrease that needs to be done in NM.

```
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 DecreasedContainerProto containers_to_decrease = 12;
}
```

2.2 Client

Client APIs need to be updated to facilitate container resource change.

2.2.1 Add public APIs in AMRMClient to facilitate the container resource change request:

We separate the resource increase and decrease APIs so that users will make a conscious decision when they initiate the requests.

```
AMRMClient.java
AMRMClientAsync.java

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

```
AMRMClientImpl.java
AMRMClientAsyncImpl.java

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

These APIs are 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 and decrease request will be cached in maps between two heartbeats (i.e., `allocate()`), and once the requests are sent, they are cleared from the maps. At any time, there can only be one increase or decrease request for a container.

For `ApplicationMaster` recovery purposes, there will also be maps to hold all pending increase and decrease requests. A pending increase/decrease request is only removed when the request is approved 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> decrease = new HashMap<>();
protected final Map<ContainerId, Resource> pendingIncrease = new HashMap<>();
protected final Map<ContainerId, Resource> pendingDecrease = new HashMap<>();
```

2.2.2 Add public APIs in NMClient to facilitate container resource increase actions:

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

```
NMClient.java
NMClientAsync.java
```

```
public abstract void increaseContainerResource(Container container);
```

```
NMClientImpl.java
NMClientAsyncImpl.java
```

```
@Override
public void increaseContainerResource(Container container);
```

2.2.3 Check container resource increase status

Successful return of the `increaseContainerResource` API does not necessarily mean that all actions related to container resource increase (such as persistence, resource monitor change, cgroup change, etc.) have been completed. Before that happens, users should not use the extra allocation (e.g., spawn more Spark tasks in the executor), otherwise, the container resource may not have been increased, and users are risking their containers to be killed by resource enforcement.

To confirm that a resource increase action has completed in NM, AM must poll NM to check container resource increase status. There are two options:

- Option 1:
  In AM, return a reference ID upon a successful call to the `increaseContainerResource` API. Then AM will call a new NMClient API (e.g., `getContainerResourceIncreaseStatus()`), passing in the reference ID along with the container ID, to check if a resource increase action on the container has completed or not.

- Option 2:
  Let `AMRMClient` cache the approved container resource increase (Obtained from the `increased_containers` set in `AllocateResponse`), and provide a local `AMRMClient` API for AM to get the latest approved container resource increase request. AM can then call the existing `getContainerStatus()` API to check if the container size in NM has been successfully changed to match the latest approved resource increase for the container.

Option 2 is preferred, because it allows reuse of the existing `getContainerStatus()` API, and does not introduce any new remote APIs.

To implement Option 2, we need to add the following:

- Update `ContainerStatusProto` protocols to add container resource so that `getContainerStatus()` API can return the capability of a container:

```
yarn_protos.proto
```

```
message ContainerStatusProto {
  optional ContainerIdProto container_id = 1;
  optional ContainerStateProto state = 2;
  optional string diagnostics = 3 [default = "N/A"];
```

```
  optional int32 exit_status = 4 [default = -1000];
  optional ResourceProto capability = 5;
}
```

- Add a local AMRMClient API to get the latest approved container resource increase:

```
AMRMClient.java
AMRMClientAsync.java

public abstract Resource getLatestApprovedResourceIncrease(ContainerId
                                                            containerId)
public abstract Resource getLatestApprovedResourceIncrease(ContainerId
                                                            containerId)
```

2.3 Resource Manager

2.3.1 Accept container resource change request:

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

Two new maps `increaseRequests` and `decreaseRequests` are introduced in the `AppSchedulingInfo` class to keep track of outstanding resource increase/decrease requests:

```
AppSchedulingInfo.java

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

In between two scheduling cycles, the following rules must apply when updating the maps:

- New increase/decrease requests only applies to containers in `RUNNING` state.
- New increase/decrease requests to the same container will always overwrite previous requests if they exist.
- At any time there can only be one increase or decrease request for a container in the maps.
- New increase/decrease requests are subject to the maximum/minimum allocation for each container defined in `yarn-default.xml`. Requests higher than the maximum will be capped to the maximum. Requests lower than the minimum will be set to the minimum.

2.3.2 Container resource increase and expiration:

During each scheduling cycle, there can only be one change request for a specific container in each scheduling cycle because of the rules set in 2.3.1.

For container resource increase, RM must handle expiration in case AM holds a container increase token but never uses it to do the increase action. 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 in Figure 3 (Normal case) and Figure 4 (Expiration case):

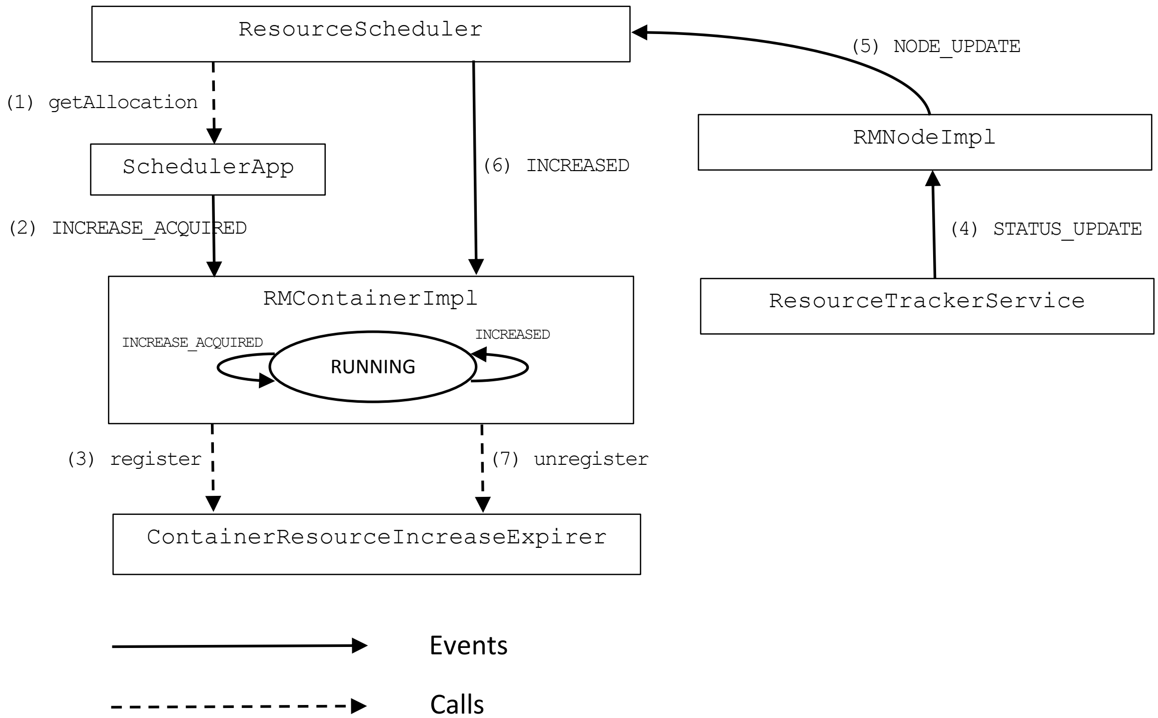


Figure 3. Container Resource Increase without expiration

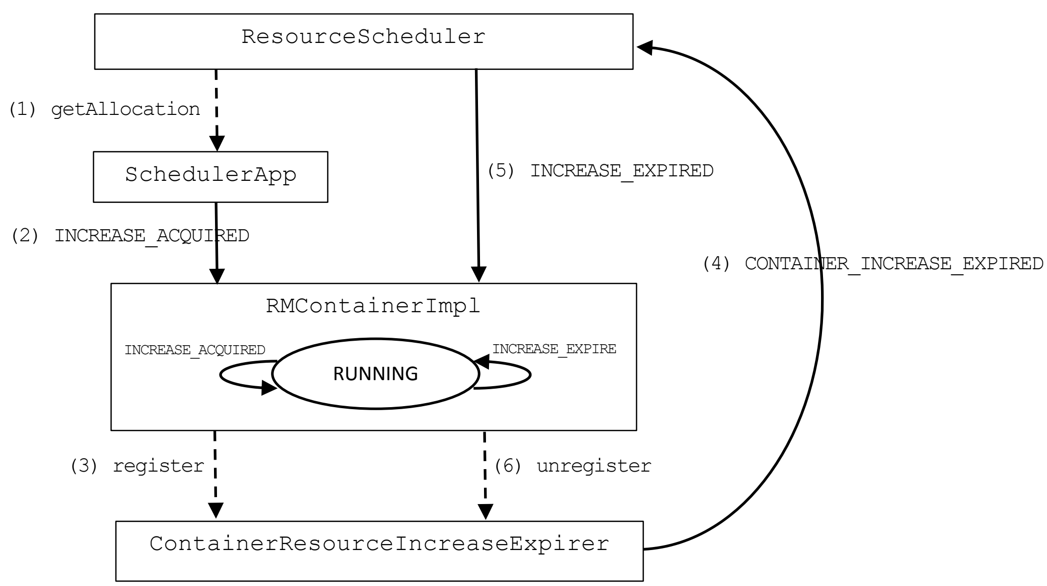


Figure 4. Container Resource Increase Expired

2.3.3 Scheduler

2.3.3.1 NodeUpdate

At each scheduler `nodeUpdate`, RM will check any unprocessed container increase messages coming from NM status update. For each message, the scheduler will do nothing except to fire an `INCREASED` event to unregister the container with `ContainerResourceIncreaseExpirer`, as illustrated in Figure 4.

2.3.3.2 Scheduling

To avoid race condition, the scheduler will skip processing of a pending resource change request of a container if there is already an approved resource increase for the same container, in particular:

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

This will guarantee that as long as there is a resource increase going on for a container in YARN, no other pending resource increase/decrease request for the same container can be processed.

Detailed scheduler design will be covered in YARN-1651.

2.4 Node Manager

2.4.1 Events and Transitions

The following new event types will be introduced to facilitate the container resource change and monitor in NM.

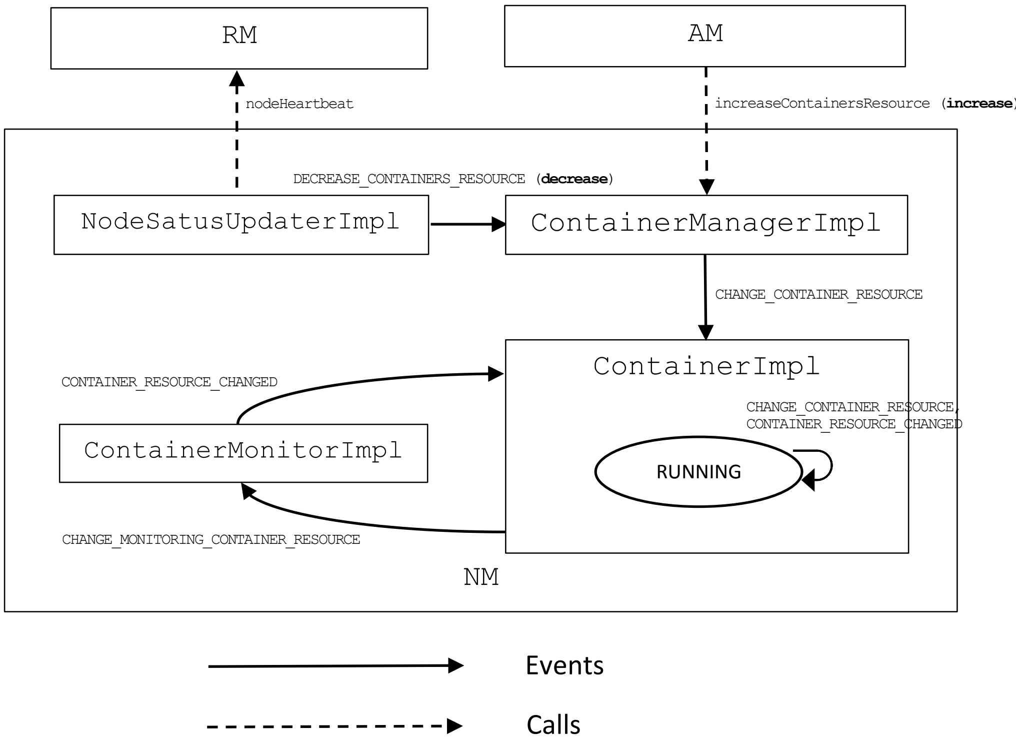
```
ContainerEventType.java
public enum ContainerEventType {

  // Producer: ContainerManager
  INIT_CONTAINER,
  KILL_CONTAINER,
  UPDATE_DIAGNOSTICS_MSG,
  CONTAINER_DONE,
  CHANGE_CONTAINER_RESOURCE,
  // Producer: ContainerMonitor
  CONTAINER_RESOURCE_CHANGED
}
```

```
ContainersMonitorEventType.java
public enum ContainersMonitorEventType {
   START_MONITORING_CONTAINER,
   STOP_MONITORING_CONTAINER,
   CHANGE_MONITORING_CONTAINER_RESOURCE
}
```

```
ContainerManagerEventType.java
public enum ContainerManagerEventType {
  FINISH_APPS,
  FINISH_CONTAINERS,
  CHANGE_CONTAINERS_RESOURCE
}
```

The container resource change logic in NodeManager is illustrated in Figure 5:



2.4.2 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.

Before Cgroup memory control is implemented, Node Manager will rely on the existing memory enforcement implemented through process tree monitoring if needed.

2.5 Recovery

2.5.1 NodeManager

Currently NodeManager only persists the initial resource capability of a container on a node (through the `RecoveredContainerState.startContainerRequest` object). With the container resize feature, we need to add the following function to facilitate the persisting of container resource changes:

```
NMStateStoreService.java
 public abstract void storeContainerResourceChange(ContainerId containerId,
   Resource capability) throws IOException;
```

```
NMLeveldbStateStoreService.java
private static final String CONTAINER_REOURCE_CHANGED_KEY_SUFFIX =
"/resourceChanged";

@Override
public void storeContainerResourceChange(ContainerId containerId,
  Resource capability) throws IOException;

public static class RecoveredContainerState {
  RecoveredContainerStatus status;
  int exitCode = ContainerExitStatus.INVALID;
  boolean killed = false;
  String diagnostics = "";
  StartContainerRequest startRequest;
  Resource capability;
  …
}
```

Every time after `ContainerImpl` receives a `CONTAINER_RESOURCE_CHANGED` event, it will call `storeContainerResourceChange` to persist the resource change in state store, and overwrites any previous resource change of the same container.

Every time a container is recovered from state store, it will be initialized with the updated resource capability.

2.5.2 ResourceManager

Nothing needs to be changed with regard to Resource Manager recovery, because Node Manager can report the correct resource capability of containers.

2.6 Topics for Discussion

2.6.1 JVM Based Container Resize

For JVM based containers, it is impossible to dynamically change the heap size of a running JVM. To use the container resize feature for a JVM based container, the Java process should be started with –Xms set to the minimum memory allocation configured for a container, and –Xmx set to the maximum memory that a container is expected to consume on a node. This may lead to some inefficiency with regard to garbage collection. In addition, there could be situation where a container has already given up certain amount of memory, but the garbage collection is delayed. In this case, downsize of the container could kill it if memory enforcement is in place. Further investigation is needed in order to understand any potential issues and to propose best practices for JVM based containers.