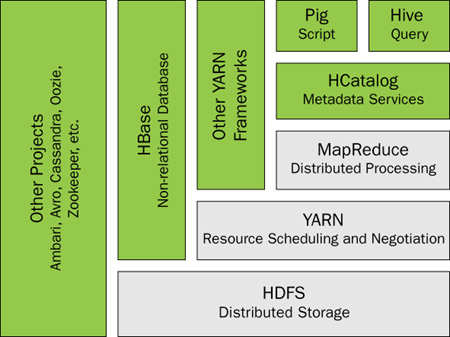
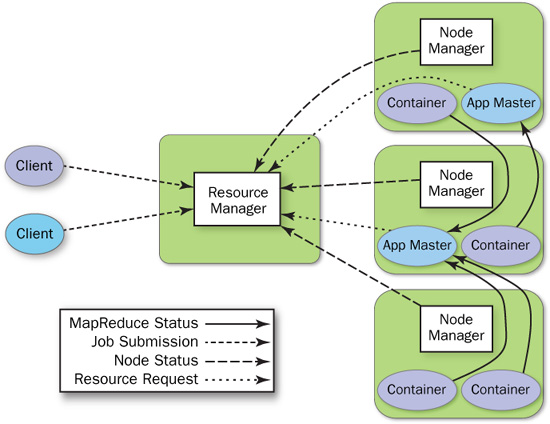
Deep Dive into Spark on YARN Resource Management

# YARN

## YARN in Hadoop Ecosystem

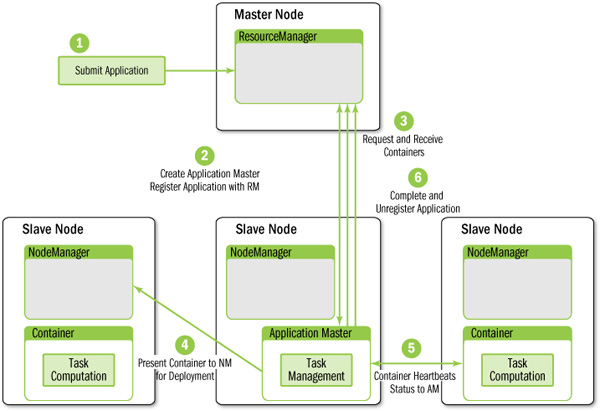


## YARN Components



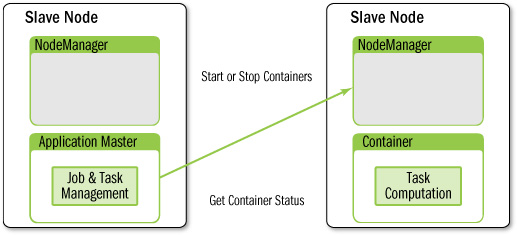
YARN Components

## Components Interactions[1]

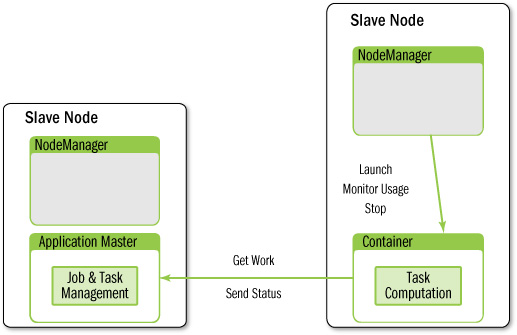


Application Master interactions

The process starts when (1) an application submits a request to the ResourceManager. Next, the ApplicationMaster is started and registers with the ResourceManager (2). The ApplicationMaster then requests containers (3) from the ResourceManager to perform actual work. The assigned containers are presented to the NodeManager for use by the ApplicationMaster (4). Computation takes place in the containers, which keep in contact (5) with the ApplicationMaster (not the ResourceManager) as the job progresses. When the application is complete, containers are stopped and the ApplicationMaster is unregistered (6) from the ResourceManager.



ApplicationMaster interacts with the NodeManagers to start/stop containers and get container status.

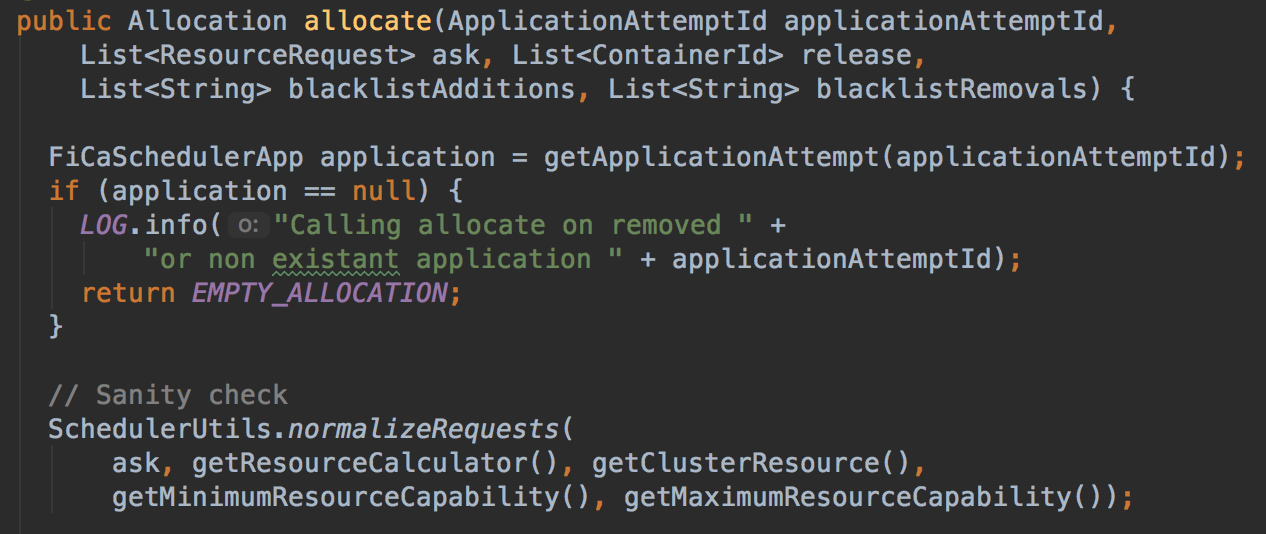


Container’s interaction with ApplicationMaster and NodeManager

## Resource Request Normalization

RM can only allocate memory to containers in increments of "yarn.scheduler.minimum-allocation-mb" and not exceed "yarn.scheduler.maximum-allocation-mb".

Utility method to normalize a list of resource requests, by insuring that the memory for each request is a multiple of minMemory and is not zero.



org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler

## Application Level Memory Allocation Aggregation

(reserved ram for container 1 \* lifetime of container 1) + (reserved ram for container 2 \* lifetime of container 2) + ... + (reserved ram for container n \* lifetime of container n)

## YARN Resource Overcommitment[3][4][5][6]

YARN Opportunistic Containers introduce in 2.9.0. As one of the future work items of opportunistic containers, overcommitment is in progress. in order to further improve the cluster resource utilization, we can schedule containers not based on the allocated resources but on the actually utilized ones. When over-committing resources, there is the risk of running out of resources in case we have an increase in the utilized resources of the already running containers. Therefore, opportunistic execution should be used for containers whose allocation goes beyond the capacity of a node. This way, we can choose opportunistic containers to kill for reclaiming resources.

## Change Resources of an Allocated Container

Currently, this is not supported.[7]

## Resource Enforcement and Isolation[8]

### Memory and CPU

Memory: monitoring container process memory usage

CPU: CGroups

### Container Monitor

Container monitor parameters

* yarn.nodemanager.container-monitor.interval-ms
* yarn.nodemanager.container-monitor.resource-calculator.class

|  |  |  |
| --- | --- | --- |
| yarn.nodemanager.container-monitor.interval-ms | - | How often to monitor containers. If not set, the value for yarn.nodemanager.resource-monitor.interval-ms will be used. If 0 or negative, container monitoring is disabled. |
| yarn.nodemanager.resource-monitor.interval-ms | 3000 | How often to monitor the node and the containers. If 0 or negative, monitoring is disabled. |

NodeManager/ContainersMonitor log example: check every 3s

|  |
| --- |
| 2018-02-11 10:14:44,200 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Starting resource-monitoring for container\_1518343391363\_0002\_01\_000001  2018-02-11 10:14:47,217 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Starting resource-monitoring for container\_1518343391363\_0002\_01\_000002  2018-02-11 10:14:50,224 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Starting resource-monitoring for container\_1518343391363\_0002\_01\_000003  2018-02-11 10:14:50,231 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3049 for container-id container\_1518343391363\_0002\_01\_000001: 353.5 MB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used  2018-02-11 10:14:50,238 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3099 for container-id container\_1518343391363\_0002\_01\_000002: 271.7 MB of 2 GB physical memory used; 2.7 GB of 4.2 GB virtual memory used  2018-02-11 10:14:50,243 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3145 for container-id container\_1518343391363\_0002\_01\_000003: 254.9 MB of 2 GB physical memory used; 2.7 GB of 4.2 GB virtual memory used  2018-02-11 10:14:53,246 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3049 for container-id container\_1518343391363\_0002\_01\_000001: 353.5 MB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used  2018-02-11 10:14:53,248 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3099 for container-id container\_1518343391363\_0002\_01\_000002: 249.8 MB of 2 GB physical memory used; 2.7 GB of 4.2 GB virtual memory used  2018-02-11 10:14:53,251 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 3145 for container-id container\_1518343391363\_0002\_01\_000003: 247.6 MB of 2 GB physical memory used; 2.7 GB of 4.2 GB virtual memory used |

NodeManager/ContainersMonitor log example: Process Tree

|  |
| --- |
| 2018-02-11 10:04:04,371 INFO org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Memory usage of ProcessTree 2511 for container-id container\_1518343391363\_0001\_02\_000001: 349.6 MB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used  2018-02-11 10:04:04,371 WARN org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Process tree for container: container\_1518343391363\_0001\_02\_000001 has processes older than 1 iteration running over the configured limit. Limit=2254857728, current usage = 2356072448  2018-02-11 10:04:04,373 WARN org.apache.hadoop.yarn.server.nodemanager.containermanager.monitor.ContainersMonitorImpl: Container [pid=2511,containerID=container\_1518343391363\_0001\_02\_000001] is running beyond virtual memory limits. Current usage: 349.6 MB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used. Killing container.  Dump of the process-tree for container\_1518343391363\_0001\_02\_000001 :  |- PID PPID PGRPID SESSID CMD\_NAME USER\_MODE\_TIME(MILLIS) SYSTEM\_TIME(MILLIS) VMEM\_USAGE(BYTES) RSSMEM\_USAGE(PAGES) FULL\_CMD\_LINE  |- 2511 2510 2511 2511 (bash) 0 0 9748480 579 /bin/bash -c /usr/java/default/bin/java -server -Xmx512m -Djava.io.tmpdir=/tmp/hadoop-admin/nm-local-dir/usercache/admin/appcache/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/tmp -Dspark.yarn.app.container.log.dir=/opt/dataplatform/hadoop-2.7.3/logs/userlogs/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001 org.apache.spark.deploy.yarn.ExecutorLauncher --arg '172.17.0.2:37753' --properties-file /tmp/hadoop-admin/nm-local-dir/usercache/admin/appcache/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/\_\_spark\_conf\_\_/\_\_spark\_conf\_\_.properties 1> /opt/dataplatform/hadoop-2.7.3/logs/userlogs/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/stdout 2> /opt/dataplatform/hadoop-2.7.3/logs/userlogs/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/stderr  |- 2516 2511 2511 2511 (java) 535 29 2346323968 88923 /usr/java/default/bin/java -server -Xmx512m -Djava.io.tmpdir=/tmp/hadoop-admin/nm-local-dir/usercache/admin/appcache/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/tmp -Dspark.yarn.app.container.log.dir=/opt/dataplatform/hadoop-2.7.3/logs/userlogs/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001 org.apache.spark.deploy.yarn.ExecutorLauncher --arg 172.17.0.2:37753 --properties-file /tmp/hadoop-admin/nm-local-dir/usercache/admin/appcache/application\_1518343391363\_0001/container\_1518343391363\_0001\_02\_000001/\_\_spark\_conf\_\_/\_\_spark\_conf\_\_.properties |

Exceeding virtual memory limits[9]

|  |
| --- |
| Current usage: 304.1 MB of 1 GB physical memory used; 2.2 GB of 2.1 GB virtual memory used. Killing container. |

This problem happens due to the combination of the following points:

* glibc>=2.10: Virtual memory footprint of processes has grown a lot since introduction of glibc:2.10 (e.g. in CentOS>=6) due to some new features; memory pool per thread.
* Java-8: The above mentioned virtual memory footprint is more pronounced in Java-8 (compared to Java-7) due to Metaspace (which has replaced PermGen).
* Yarn: Yarn kills process that exceeds the limits that has been set for the resources it monitors, and Yarn by default monitors virtual memory.

We can resolve the problem by eliminating one or some of the above mentioned points. E.g. turn off Yarn vmem check.

|  |
| --- |
| <property>  <name>yarn.nodemanager.vmem-check-enabled</name>  <value>false</value>  </property> |

### Container Metrics[10][11][12][13]

Container metrics parameters

* yarn.nodemanager.container-metrics.enable
* yarn.nodemanager.container-metrics.period-ms

Metrics for container's actual memory utilization are implemented since 2.7.0, however documented since 2.9.0.

* PMemUsageMBsMaxMBs: Maximum physical memory used in MB
* pMemLimitMBs: Physical memory limit of the container in MB

Container metrics published via JMX

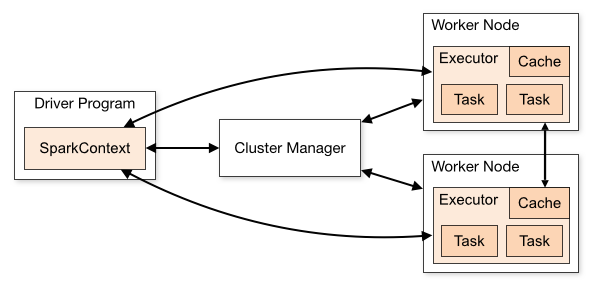
|  |
| --- |
| curl -i http://localhost:8042/jmx?qry=Hadoop:service=NodeManager,name=ContainerResource\_container\*  {  "beans" : [ {  "name" : "Hadoop:service=NodeManager,name=ContainerResource\_container\_1519438765876\_0003\_01\_000003",  "modelerType" : "ContainerResource\_container\_1519438765876\_0003\_01\_000003",  "tag.ContainerResource" : "container\_1519438765876\_0003\_01\_000003",  "tag.Context" : "container",  "tag.ContainerPid" : "5162",  "tag.Hostname" : "707730f8893a",  "PMemUsageMBsNumUsage" : 1921,  "PMemUsageMBsAvgMBs" : 278.31203801478347,  "PMemUsageMBsStdevMBs" : 3.033932549667687,  "PMemUsageMBsIMinMBs" : 272.0,  "PMemUsageMBsIMaxMBs" : 289.0,  "PMemUsageMBsMinMBs" : 104.0,  "PMemUsageMBsMaxMBs" : 289.0,  "PCpuUsagePercentNumUsage" : 1920,  "PCpuUsagePercentAvgPercents" : 0.005807814149947192,  "PCpuUsagePercentStdevPercents" : 0.15407154923732722,  "PCpuUsagePercentIMinPercents" : 0.0,  "PCpuUsagePercentIMaxPercents" : 5.0,  "PCpuUsagePercentMinPercents" : 0.0,  "PCpuUsagePercentMaxPercents" : 93.0,  "MilliVcoreUsageNumUsage" : 1920,  "MilliVcoreUsageAvgMilliVcores" : 2.143083421330515,  "MilliVcoreUsageStdevMilliVcores" : 4.445135662498212,  "MilliVcoreUsageIMinMilliVcores" : 0.0,  "MilliVcoreUsageIMaxMilliVcores" : 112.0,  "MilliVcoreUsageMinMilliVcores" : 0.0,  "MilliVcoreUsageMaxMilliVcores" : 1879.0,  "pMemLimitMBs" : 2048,  "vMemLimitMBs" : 4300,  "vCoreLimit" : 1  }, {  "name" : "Hadoop:service=NodeManager,name=ContainerResource\_container\_1519438765876\_0003\_01\_000002",  "modelerType" : "ContainerResource\_container\_1519438765876\_0003\_01\_000002",  "tag.ContainerResource" : "container\_1519438765876\_0003\_01\_000002",  "tag.Context" : "container",  "tag.ContainerPid" : "5138",  "tag.Hostname" : "707730f8893a",  "PMemUsageMBsNumUsage" : 1921,  "PMemUsageMBsAvgMBs" : 260.4466737064415,  "PMemUsageMBsStdevMBs" : 12.545534897985252,  "PMemUsageMBsIMinMBs" : 242.0,  "PMemUsageMBsIMaxMBs" : 276.0,  "PMemUsageMBsMinMBs" : 163.0,  "PMemUsageMBsMaxMBs" : 276.0,  "PCpuUsagePercentNumUsage" : 1920,  "PCpuUsagePercentAvgPercents" : 0.008975712777191135,  "PCpuUsagePercentStdevPercents" : 0.2629100956114575,  "PCpuUsagePercentIMinPercents" : 0.0,  "PCpuUsagePercentIMaxPercents" : 9.0,  "PCpuUsagePercentMinPercents" : 0.0,  "PCpuUsagePercentMaxPercents" : 30.0,  "MilliVcoreUsageNumUsage" : 1920,  "MilliVcoreUsageAvgMilliVcores" : 2.1763463569165777,  "MilliVcoreUsageStdevMilliVcores" : 6.31749766224844,  "MilliVcoreUsageIMinMilliVcores" : 0.0,  "MilliVcoreUsageIMaxMilliVcores" : 186.0,  "MilliVcoreUsageMinMilliVcores" : 0.0,  "MilliVcoreUsageMaxMilliVcores" : 607.0,  "pMemLimitMBs" : 2048,  "vMemLimitMBs" : 4300,  "vCoreLimit" : 1  }, {  "name" : "Hadoop:service=NodeManager,name=ContainerResource\_container\_1519438765876\_0003\_01\_000001",  "modelerType" : "ContainerResource\_container\_1519438765876\_0003\_01\_000001",  "tag.ContainerResource" : "container\_1519438765876\_0003\_01\_000001",  "tag.Context" : "container",  "tag.ContainerPid" : "5078",  "tag.Hostname" : "707730f8893a",  "PMemUsageMBsNumUsage" : 1922,  "PMemUsageMBsAvgMBs" : 345.4746568109814,  "PMemUsageMBsStdevMBs" : 2.1215424034259986,  "PMemUsageMBsIMinMBs" : 339.0,  "PMemUsageMBsIMaxMBs" : 347.0,  "PMemUsageMBsMinMBs" : 187.0,  "PMemUsageMBsMaxMBs" : 354.0,  "PCpuUsagePercentNumUsage" : 1921,  "PCpuUsagePercentAvgPercents" : 0.0026399155227032787,  "PCpuUsagePercentStdevPercents" : 0.11488941471483062,  "PCpuUsagePercentIMinPercents" : 0.0,  "PCpuUsagePercentIMaxPercents" : 5.0,  "PCpuUsagePercentMinPercents" : 0.0,  "PCpuUsagePercentMaxPercents" : 73.0,  "MilliVcoreUsageNumUsage" : 1921,  "MilliVcoreUsageAvgMilliVcores" : 2.0385427666314695,  "MilliVcoreUsageStdevMilliVcores" : 3.921185120752089,  "MilliVcoreUsageIMinMilliVcores" : 0.0,  "MilliVcoreUsageIMaxMilliVcores" : 119.0,  "MilliVcoreUsageMinMilliVcores" : 0.0,  "MilliVcoreUsageMaxMilliVcores" : 1470.0,  "pMemLimitMBs" : 1024,  "vMemLimitMBs" : 2150,  "vCoreLimit" : 1  } ] |

### Metrics 2.0 Framework[14]

The framework provides a variety of ways to implement metrics instrumentation easily via the simple MetricsSource interface or the even simpler and more concise and declarative metrics annotations. The consumers of metrics just need to implement the simple MetricsSink interface. Producers register the metrics sources with a metrics system, while consumers register the sinks. A default metrics system is provided to marshal metrics from sources to sinks based on (per source/sink) configuration options. All the metrics are also published and queryable via the standard JMX MBean interface.

# Spark Application Deploying

## Spark Application Model and Deploying[15]



Unit of Computation:

* Application
* Job
* Stage
* Task

Cluster Manager

* Standalone
* YARN
* Mesos
* Kubernetes

## Resource Allocation

Parameters:

* Spark.executor.memory
* Spark.executor.cores
* Spark.driver.memory
* spark.driver.cores

## Dynamic Resource Allocation[16]

### Background

Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. This means that your application may give resources back to the cluster if they are no longer used and request them again later when there is demand. This feature is disabled by default and available on all coarse-grained cluster managers, i.e. standalone mode, YARN mode, and Mesos coarse-grained mode. This feature is introduced since Spark 1.3.

There are two requirements for using this feature. First, your application must set spark.dynamicAllocation.enabled to true. Second, you must set up an external shuffle service on each worker node in the same cluster and set spark.shuffle.service.enabled to true in your application. The purpose of the external shuffle service is to allow executors to be removed without deleting shuffle files written by them.

We're blocked on YARN-1197 for dynamically changing the resources within executors.[17]

At a high level, Spark should relinquish executors when they are no longer used and acquire executors when they are needed. Since there is no definitive way to predict whether an executor that is about to be removed will run a task in the near future, or whether a new executor that is about to be added will actually be idle, we need a set of heuristics to determine when to remove and request executors.

### Request Policy

A Spark application with dynamic allocation enabled requests additional executors when it has pending tasks waiting to be scheduled. This condition necessarily implies that the existing set of executors is insufficient to simultaneously saturate all tasks that have been submitted but not yet finished.

Spark requests executors in rounds. The actual request is triggered when there have been pending tasks for spark.dynamicAllocation.schedulerBacklogTimeout seconds, and then triggered again every spark.dynamicAllocation.sustainedSchedulerBacklogTimeout seconds thereafter if the queue of pending tasks persists. Additionally, the number of executors requested in each round increases exponentially from the previous round. For instance, an application will add 1 executor in the first round, and then 2, 4, 8 and so on executors in the subsequent rounds.

The motivation for an exponential increase policy is twofold. First, an application should request executors cautiously in the beginning in case it turns out that only a few additional executors is sufficient. This echoes the justification for TCP slow start. Second, the application should be able to ramp up its resource usage in a timely manner in case it turns out that many executors are actually needed.

### Remove Policy

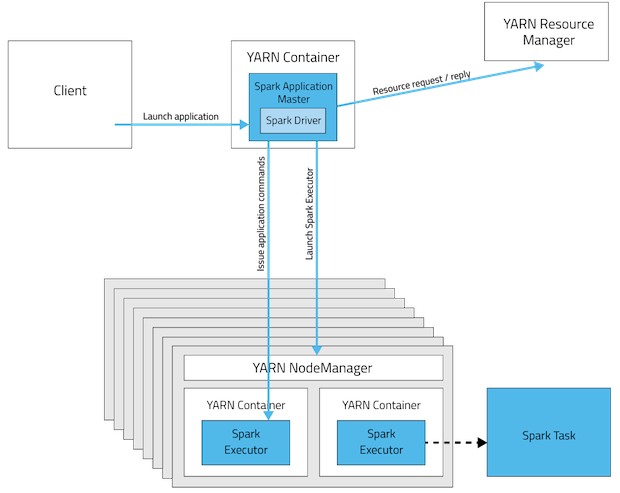
The policy for removing executors is much simpler. A Spark application removes an executor when it has been idle for more than spark.dynamicAllocation.executorIdleTimeout seconds. Note that, under most circumstances, this condition is mutually exclusive with the request condition, in that an executor should not be idle if there are still pending tasks to be scheduled.

## Coarse-grained vs Fine-grained

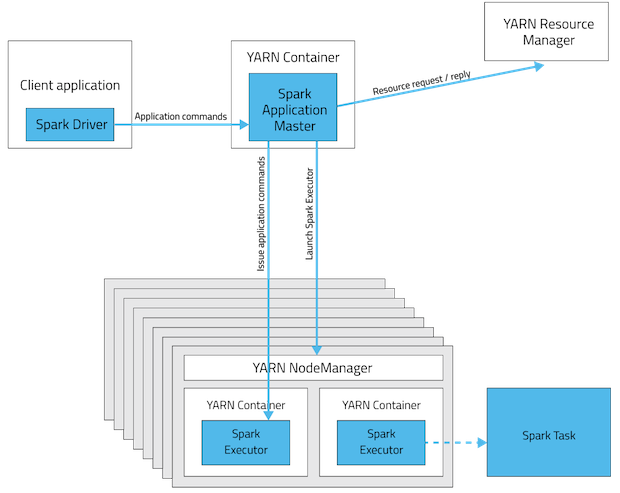
Mesos is the only cluster manager that had a fine-grained mode, but it's more often than not problematic, and it's a maintenance burden. It was removed from 2.0 release. Similar functionality can be achieved with dynamic allocation + coarse-grained mode.[18]

# Spark on YARN

## Spark on YARN Deploy Mode[19]

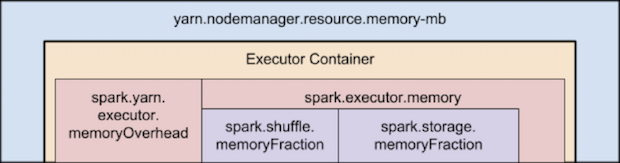


yarn-cluster mode



yarn-client mode

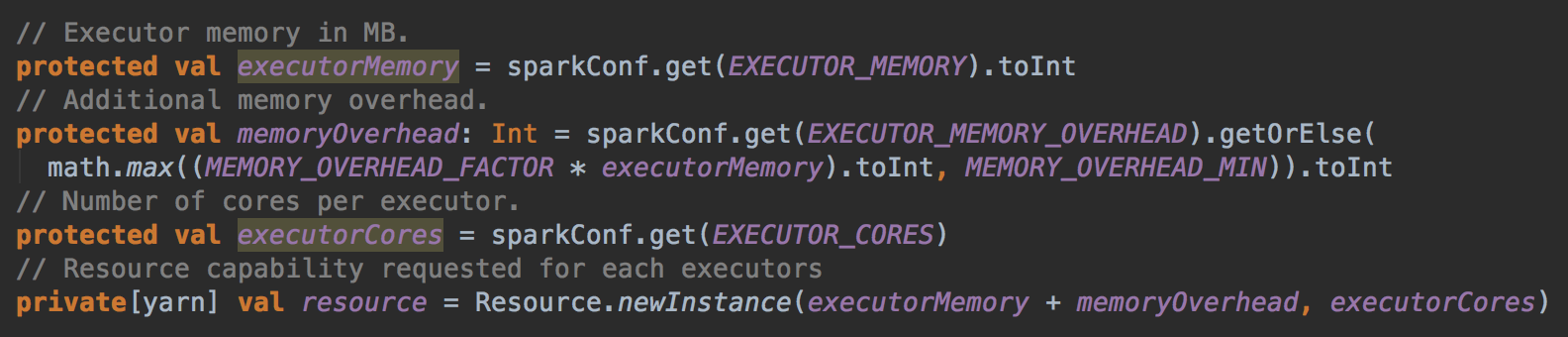
## Resource Request



Memory Hierarchy in Spark on YARN[20]

Parameters:

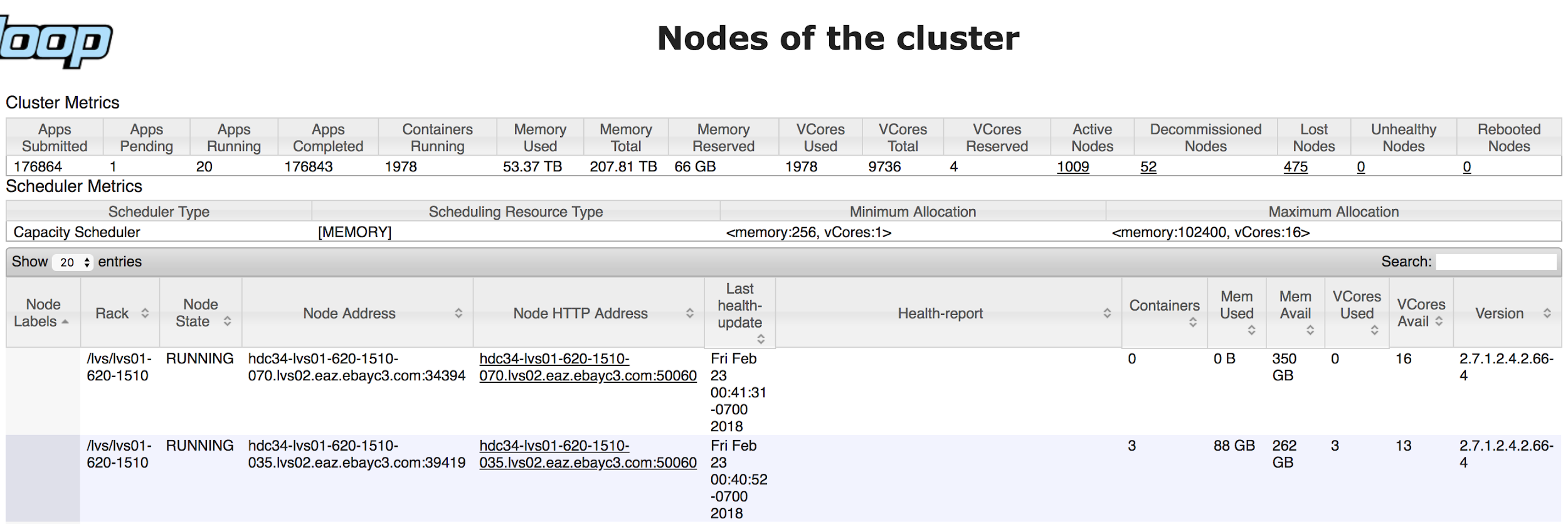
* spark.yarn.executor.memoryOverhead
* spark.yarn.driver.memoryOverhead
* spark.yarn.am.memoryOverhead



org.apache.spark.deploy.yarn.YarnAllocator

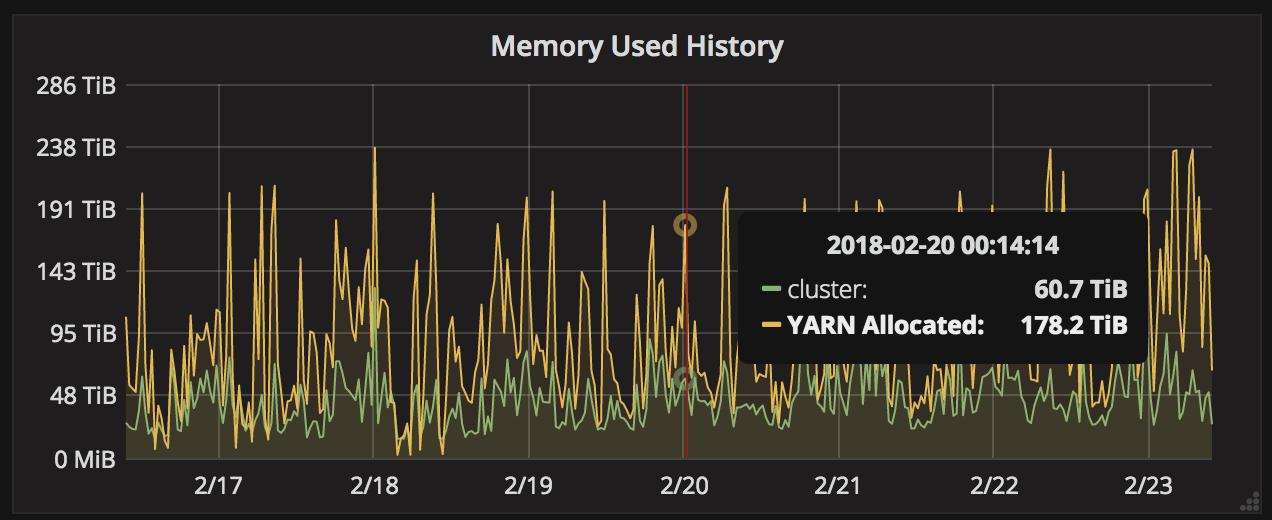
# Hercules

## Nodes

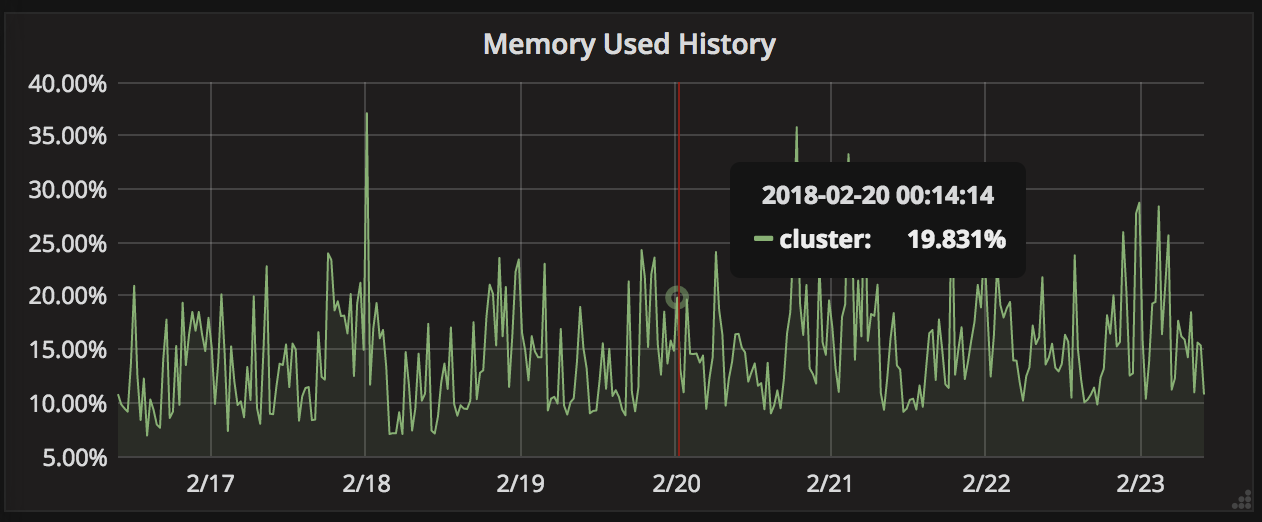


https://hercules-lvs-rm-1.vip.ebay.com:50030/cluster/nodes

## Memory Usage



Utilized (green) vs Allocated (yellow)



Utilized Ratio

See more metrics at:

http://hms021.stratus.lvs.ebay.com:3000/dashboard/db/cluster-aggragated-overview?orgId=1&from=now-7d&to=now&var-cluster=hercules&var-rack=All&var-zone=All&var-vpc=All

## NodeManager Configuration

|  |
| --- |
| <property>  <name>yarn.nodemanager.vmem-check-enabled</name>  <value>true</value>  <source>yarn-default.xml</source>  </property>  <property>  <name>yarn.nodemanager.vmem-pmem-ratio</name>  <value>4</value>  <source>yarn-site.xml</source>  </property> |

See more configuration of one of NodeManagers, e.g.

https://hdc34-lvs01-620-1501-027.lvs02.eaz.ebayc3.com:50060/conf

# References

[1] Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2. Arun Murthy.

[2] YARN-415: Capture aggregate memory allocation at the app-level for chargeback.

[3] http://hadoop.apache.org/docs/r2.9.0/hadoop-yarn/hadoop-yarn-site/OpportunisticContainers.html

[4] https://www.slideshare.net/HadoopSummit/investing-the-effects-of-overcommitting-yarn-resources

[5] YARN-5202: Dynamic Overcommit of Node Resources - POC.

[6] YARN-1011: [Umbrella] Schedule containers based on utilization of currently allocated containers.

[7] YARN-1197: Support changing resources of an allocated container.

[8] https://hortonworks.com/blog/managing-cpu-resources-in-your-hadoop-yarn-clusters/

[9] https://a-ghorbani.github.io/2016/12/23/spark-on-yarn-and-java-8-and-virtual-memory-error

[10] YARN-2984: Metrics for container's actual memory usage

[11] YARN-3122: Metrics for container's actual CPU usage

[12] YARN-6028: Add document for container metrics

[13] http://hadoop.apache.org/docs/r2.9.0/hadoop-project-dist/hadoop-common/Metrics.html#ContainerMetrics

[14] http://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/metrics2/package-summary.html

[15] https://spark.apache.org/docs/latest/cluster-overview.html

[16] https://spark.apache.org/docs/latest/job-scheduling.html#dynamic-resource-allocation

[17] SPARK-3174: Provide elastic scaling within a Spark application. This is also known as dynamic allocation.

[18] SPARK-11857: Remove Mesos fine-grained mode subject to discussions.

[19] http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/

[20] https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/