

Hortonworks Data Platform

Using Apache Hadoop

(Jul 2, 2014)

Hortonworks Data Platform: Using Apache Hadoop

Copyright © 2012, 2013 Hortonworks, Inc. Some rights reserved.

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, Zookeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, [training](#) and partner-enablement services. All of our technology is, and will remain free and open source.

Please visit the [Hortonworks Data Platform](#) page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the [Support](#) or [Training](#) page. Feel free to [Contact Us](#) directly to discuss your specific needs.



Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 3.0 License.
<http://creativecommons.org/licenses/by-sa/3.0/legalcode>

Table of Contents

1. Using Apache Hadoop	1
1.1. Hadoop Common	1
1.2. Using Hadoop HDFS	1
1.3. Using Hadoop MapReduce on YARN	2
1.3.1. Running MapReduce on Hadoop YARN	2
1.3.2. MapReduce Version 2 Troubleshooting Guide	18
1.3.3. YARN Components	45
1.3.4. Using Hadoop YARN	52

1. Using Apache Hadoop

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

The project includes these modules:

- Hadoop Common: The set of common utilities that support other Hadoop modules
- Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data
- Hadoop YARN: Framework for job scheduling and cluster resource management
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets

To learn more about Apache Hadoop, see [Apache Hadoop](#).

In this section:

- [Hadoop Common](#)
- [Using Hadoop HDFS](#)
- [Using Hadoop MapReduce on YARN](#)
- [Using Hadoop YARN](#)

1.1. Hadoop Common

Hadoop Common is the set of common utilities that support other Hadoop modules.

In this section:

- [Setting up a Single Node Cluster](#)
- [Cluster Setup](#)
- [CLI MiniCluster](#)
- [Using File System \(FS\) Shell](#)
- [Hadoop Commands Reference](#)

1.2. Using Hadoop HDFS

HDFS is a distributed file system that provides high-throughput access to data. It provides a limited interface for managing the file system to allow it to scale and provide high throughput. HDFS creates multiple replicas of each data block and distributes them on computers throughout a cluster to enable reliable and rapid access.

Use the following resources to learn more about Hadoop HDFS:

- [HDFS High Availability Using the Quorum Journal Manager](#)
- [HDFS High Availability with NFS](#)
- [WebHDFS REST API](#)

1.3. Using Hadoop MapReduce on YARN

In Hadoop version 1, MapReduce was responsible for both processing and cluster resource management. In Apache Hadoop version 2, cluster resource management has been moved from MapReduce into YARN, thus enabling other application engines to utilize YARN and Hadoop, while also improving the performance of MapReduce.

Use the following resources to learn more about using Hadoop MapReduce on YARN:

- [Running MapReduce on Hadoop YARN](#)
- [MapReduce Version 2 Troubleshooting Guide](#)
- [YARN Components](#)
- [YARN Capacity Scheduler](#)
- [Encrypted Shuffle](#)
- [Pluggable Shuffle and Pluggable Sort](#)

1.3.1. Running MapReduce on Hadoop YARN

The introduction of a Hadoop Version 2 has changed the way MapReduce applications run on a cluster. Unlike the monolithic MapReduce-Schedule in Hadoop Version 1, Hadoop YARN has generalized the cluster resources available to users. To ensure compatibility with Hadoop Version 1, the YARN team has written a MapReduce framework that works on top of YARN. The framework is highly compatible with Hadoop Version 1, with only a small number of issues to consider. As with Hadoop Version 1, Hadoop YARN includes virtually the same MapReduce examples and benchmarks that help demonstrate how Hadoop YARN functions.

Use the following resources to learn more about Running MapReduce on YARN:

- [Running MapReduce Examples on Hadoop YARN](#)
- [MapReduce Compatibility](#)
- [The MapReduce Application Master](#)
- [Calculating the Capacity of a Node](#)
- [Changes to the Shuffle Service](#)
- [Running Existing Hadoop Version 1 Applications on YARN](#)
- [Running Existing Version 1 Code on YARN](#)
- [Uber Jobs \(Technical Preview\)](#)

1.3.1.1. Running MapReduce Examples on Hadoop YARN

The MapReduce examples are located in `hadoop-[VERSION]/share/hadoop/mapreduce`. Depending on where you installed Hadoop, this path may vary. For the purposes of this example let's define:

```
export YARN_EXAMPLES=$YARN_HOME/share/hadoop/mapreduce
```

`$YARN_HOME` should be defined as part of your installation. Also, the following examples have a version tag, in this case "2.1.0-beta." Your installation may have a different version tag.

The following sections provide some examples of Hadoop YARN programs and benchmarks.

Listing Available Examples

Using our `$YARN_HOME` environment variable, we can get a list of the available examples by running:

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.1.0-beta.jar
```

This command returns a list of the available examples:

```
An example program must be given as the first argument.
Valid program names are:
  aggregatewordcount: An Aggregate-based map/reduce program that counts
the words in the input files.
  aggregatewordhist: An Aggregate-based map/reduce program that computes
the histogram of the words in the input files.
  bbbp: A map/reduce program that uses Bailey-Borwein-Plouffe to compute
exact digits of Pi.
  dbcount: An example job that counts the pageview counts from a database.
  distbbp: A map/reduce program that uses a BBP-type formula to compute
exact bits of Pi.
  grep: A map/reduce program that counts the matches of a regex in the
input.
  join: A job that effects a join over sorted, equally partitioned
datasets.
  multifilewc: A job that counts words from several files.
  pentomino: A map/reduce tile-laying program that finds solutions to
pentomino problems.
  pi: A map/reduce program that estimates Pi using a quasi-Monte Carlo
method.
  randomtextwriter: A map/reduce program that writes 10GB of random
textual data per node.
  randomwriter: A map/reduce program that writes 10GB of random data per
node.
  secondarysort: An example defining a secondary sort to the reduce.
  sort: A map/reduce program that sorts the data written by the random
writer.
  sudoku: A sudoku solver.
  teragen: Generate data for the terasort.
  terasort: Run the terasort.
  teravalidate: Check the results of the terasort.
  wordcount: A map/reduce program that counts the words in the input
files.
  wordmean: A map/reduce program that counts the average length of the
```

```
words in the input files.
wordmedian: A map/reduce program that counts the median length of the
words in the input files.
wordstandarddeviation: A map/reduce program that counts the standard
deviation of the length of the words in the input files.
```

To illustrate several features of Hadoop YARN, we will show you how to run the `pi` and `terasort` examples, as well as the `TestDFSIO` benchmark.

Running the `pi` Example

To run the `pi` example with 16 maps and 100000 samples, run the following command:

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.1.0-beta.jar pi 16 100000
```

This command should return the following result (after the Hadoop messages):

```
13/10/14 20:10:01 INFO mapreduce.Job: map 0% reduce 0%
13/10/14 20:10:08 INFO mapreduce.Job: map 25% reduce 0%
13/10/14 20:10:16 INFO mapreduce.Job: map 56% reduce 0%
13/10/14 20:10:17 INFO mapreduce.Job: map 100% reduce 0%
13/10/14 20:10:17 INFO mapreduce.Job: map 100% reduce 100%
13/10/14 20:10:17 INFO mapreduce.Job: Job job_1381790835497_0003 completed
successfully
13/10/14 20:10:17 INFO mapreduce.Job: Counters: 44
    File System Counters
        FILE: Number of bytes read=358
        FILE: Number of bytes written=1365080
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=4214
        HDFS: Number of bytes written=215
        HDFS: Number of read operations=67
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=3
    Job Counters
        Launched map tasks=16
        Launched reduce tasks=1
        Data-local map tasks=14
        Rack-local map tasks=2
        Total time spent by all maps in occupied slots (ms)=174725
        Total time spent by all reduces in occupied slots
        (ms)=7294
    Map-Reduce Framework
        Map input records=16
        Map output records=32
        Map output bytes=288
        Map output materialized bytes=448
        Input split bytes=2326
        Combine input records=0
        Combine output records=0
        Reduce input groups=2
        Reduce shuffle bytes=448
        Reduce input records=32
        Reduce output records=0
        Spilled Records=64
        Shuffled Maps =16
        Failed Shuffles=0
        Merged Map outputs=16
```

```

GC time elapsed (ms)=195
CPU time spent (ms)=7740
Physical memory (bytes) snapshot=6143696896
Virtual memory (bytes) snapshot=23140454400
Total committed heap usage (bytes)=4240769024

Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0

File Input Format Counters
  Bytes Read=1888
File Output Format Counters
  Bytes Written=97
Job Finished in 20.854 seconds
Estimated value of Pi is 3.14127500000000000000

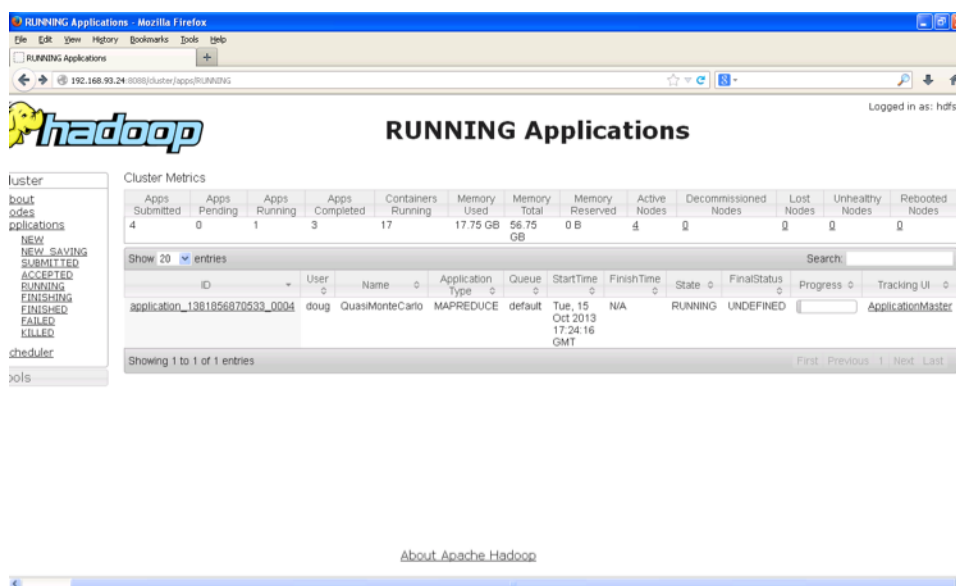
```

Note that the MapReduce progress is shown – as is the case with MapReduce V1 – but the application statistics are different. Most of the statistics are self-explanatory. The one important item to note is that the YARN “Map-Reduce Framework” is used to run the program. The use of this framework, which is designed to be compatible with Hadoop V1, will be discussed further in subsequent sections.

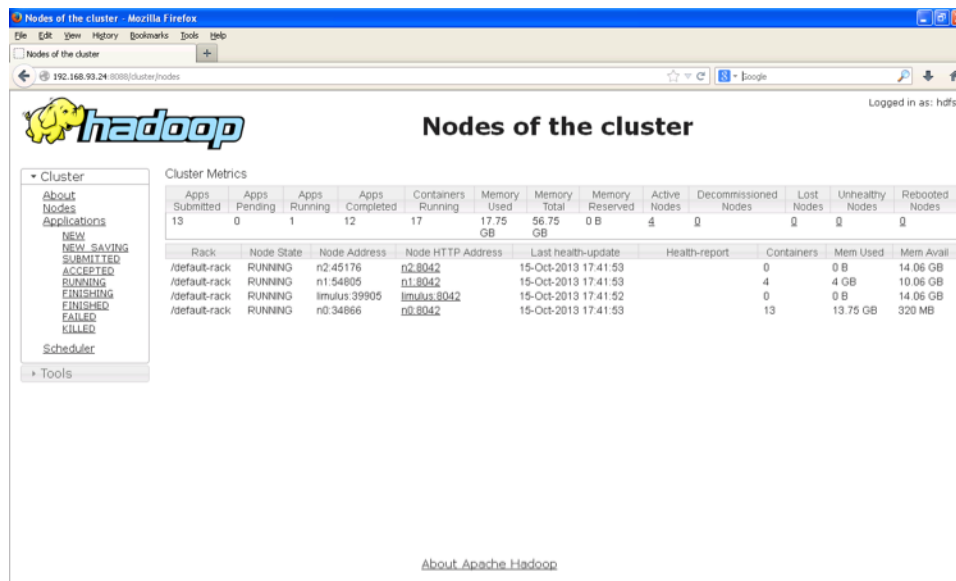
Using the Web GUI to Monitor Examples

The Hadoop YARN web Graphical User Interface (GUI) has been updated for Hadoop Version 2. This section shows you how to use the web GUI to monitor and find information about YARN jobs. In the following examples, we use the `pi` application, which can run quickly and be finished before you have explored the GUI. A longer running application – such as `terasort` – may be helpful when exploring all the various links in the GUI.

The following figure shows the main YARN web interface (<http://hostname:8088>).



If you look at the Cluster Metrics table, you will see some new information. First, you will notice that rather than Hadoop Version 1 “Map/Reduce Task Capacity,” there is now information on the number of running Containers. If YARN is running a MapReduce job, these Containers will be used for both map and reduce tasks. Unlike Hadoop Version 1, in Hadoop Version 2 the number of mappers and reducers is not fixed. There are also memory metrics and a link to node status. To display a summary of the node activity, click **Nodes**. The following image shows the node activity while the pi application is running. Note again the number of Containers, which are used by the MapReduce framework as either mappers or reducers.



Nodes of the cluster - Mozilla Firefox

Nodes of the cluster

192.168.93.24:8080/cluster/nodes

Logged in as: hdfs

hadoop

Nodes of the cluster

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
13	0	1	12	17	17.75 GB	56.75 GB	0 B	4	0	0	0	0

Rack	Node State	Node Address	Node HTTP Address	Last health-update	Health-report	Containers	Mem Used	Mem Avail
/default-rack	RUNNING	n2-45176	n2-8042	15-Oct-2013 17:41:53		0	0 B	14.06 GB
/default-rack	RUNNING	n1-54805	n1-8042	15-Oct-2013 17:41:53		4	4 GB	10.06 GB
/default-rack	RUNNING	imulus-39905	imulus-8042	15-Oct-2013 17:41:52		0	0 B	14.06 GB
/default-rack	RUNNING	n0-34866	n0-8042	15-Oct-2013 17:41:53		13	13.75 GB	320 MB

[About Apache Hadoop](#)

If you navigate back to the main Running Applications window and click the `application_138...` link, the Application status page appears. This page provides information similar to that on the Running Applications page, but only for the selected job.

Application Overview

User: doug
 Name: QuasiMonteCarlo
 Application Type: MAPREDUCE
 State: RUNNING
 FinalStatus: UNDEFINED
 Started: 15-Oct-2013 13:24:16
 Elapsed: 7sec
 Tracking URL: [ApplicationMaster](#)
 Diagnostics:

Attempt Number	Start Time	Node	Logs
1	15-Oct-2013 13:24:16	n0:8042	logs

[About Apache Hadoop](#)

Clicking the **ApplicationMaster** link on the Application status page opens the MapReduce Application page shown in the following figure. Note that the link to the ApplicationMaster is also on the main Running Applications screen in the last column.

MapReduce Application
 application_1381856870533_0004

Active Jobs

Show 20 entries

Job ID	Name	State	Map Progress	Maps Total	Maps Completed	Reduce Progress	Reduces Total	Reduces Completed
job_1381856870533_0004	QuasiMonteCarlo	RUNNING		16	2		1	0

Showing 1 to 1 of 1 entries

[About Apache Hadoop](#)

Details about the MapReduce process can be observed on the MapReduce Application page. Instead of Containers, the MapReduce application now refers to Maps and Reduces. Clicking the `job_138...` link opens the MapReduce Job page:

MapReduce Job
job_1381856870533_0004

Job Overview

Job Name: QuasiMonteCarlo
State: RUNNING
Uberized: false
Started: Tue Oct 15 13:24:20 EDT 2013
Elapsed: 14sec

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Oct 15 13:24:18 EDT 2013	n0:8042	logs

Task Type	Progress	Total	Pending	Running	Complete
Map	<div></div>	16	0	14	2
Reduce	<div></div>	1	0	1	0

Attempt Type	New	Running	Failed	Killed	Successful
Maps	0	14	0	0	2
Reduces	0	1	0	0	0

[About Apache Hadoop](#)

The MapReduce Job page provides more detail about the status of the job. When the job is finished, the page is updated as shown in the following figure:

job_1381856870533_0004

Job Overview

Job Name: QuasiMonteCarlo
User Name: doug
Queue: default
State: SUCCEEDED
Uberized: false
Started: Tue Oct 15 13:24:20 EDT 2013
Finished: Tue Oct 15 13:24:41 EDT 2013
Elapsed: 20sec

Diagnostics:

Average Map Time: 15sec
Average Reduce Time: 0sec
Average Shuffle Time: 13sec
Average Merge Time: 0sec

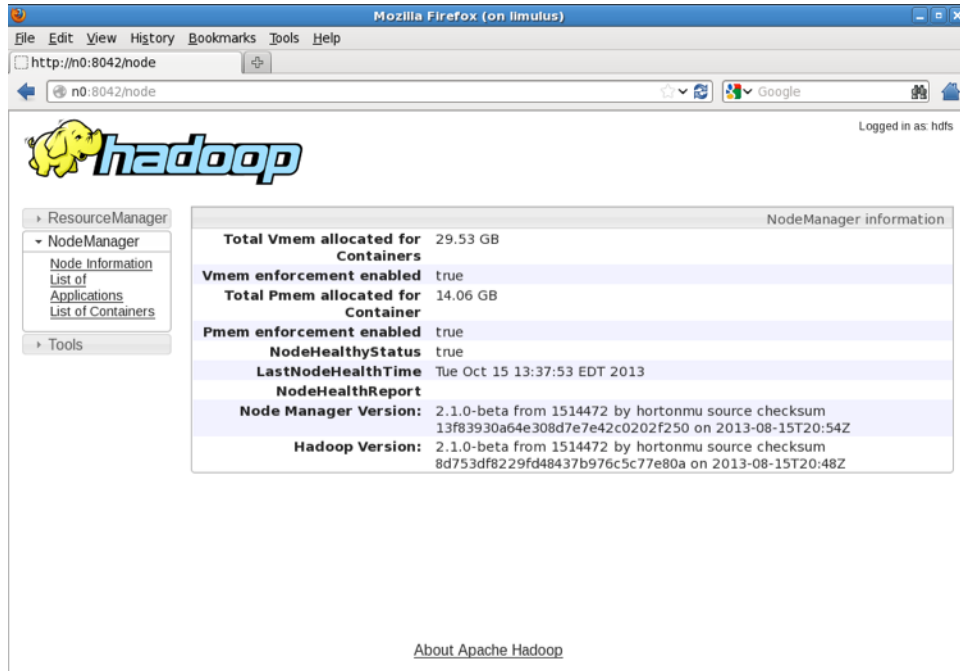
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Oct 15 13:24:18 EDT 2013	n0:8042	logs

Task Type	Total	Complete
Map	16	16
Reduce	1	1

Attempt Type	Failed	Killed	Successful
Maps	0	0	16
Reduces	0	0	1

If you click the Node used to run the ApplicationMaster (n0:8042 above), a NodeManager summary page appears, as shown in the following figure. Again, the NodeManager only tracks Containers. The actual tasks that the Containers run is determined by the ApplicationMaster.

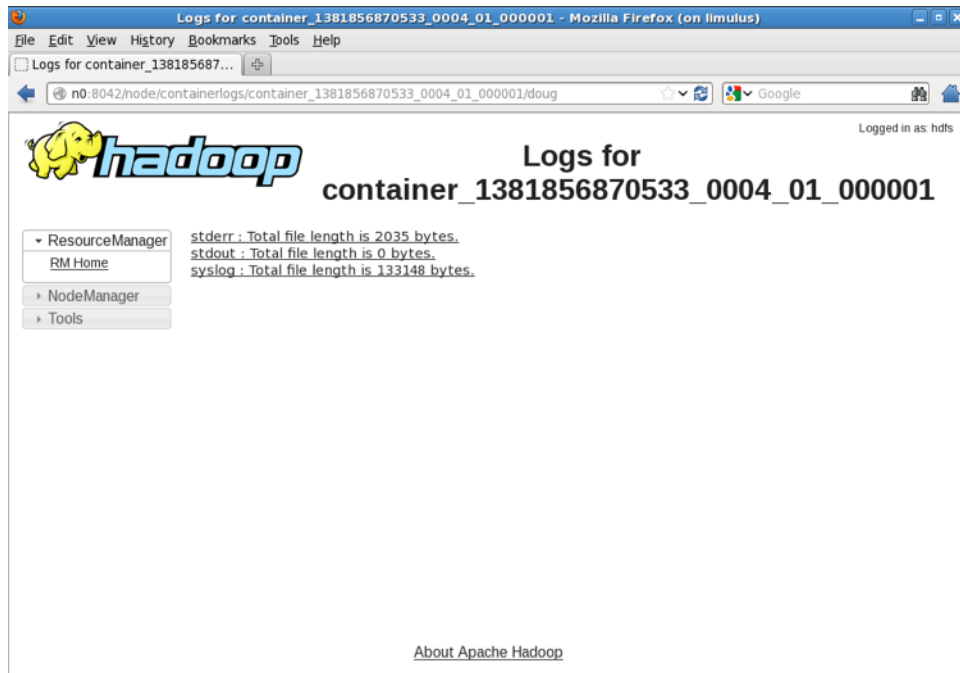


The screenshot shows a Mozilla Firefox browser window with the address bar displaying `http://n0:8042/node`. The page title is "NodeManager information". The Hadoop logo is visible in the top left. The page is logged in as "hdfs". A sidebar on the left contains links for "ResourceManager", "NodeManager", "Node information", "List of Applications", "List of Containers", and "Tools". The main content area displays the following information:

Total Vmem allocated for Containers	29.53 GB
Vmem enforcement enabled	true
Total Pmem allocated for Container	14.06 GB
Pmem enforcement enabled	true
NodeHealthyStatus	true
LastNodeHealthTime	Tue Oct 15 13:37:53 EDT 2013
NodeHealthReport	
Node Manager Version:	2.1.0-beta from 1514472 by hortonmu source checksum 13f83930a64e308d7e7e42c0202f250 on 2013-08-15T20:54Z
Hadoop Version:	2.1.0-beta from 1514472 by hortonmu source checksum 8d753df8229fd48437b976c5c77e80a on 2013-08-15T20:48Z

At the bottom of the page, there is a link for "About Apache Hadoop".

If you navigate back to the MapReduce Job page, you can access log files for the ApplicationMaster by clicking the **logs** link:



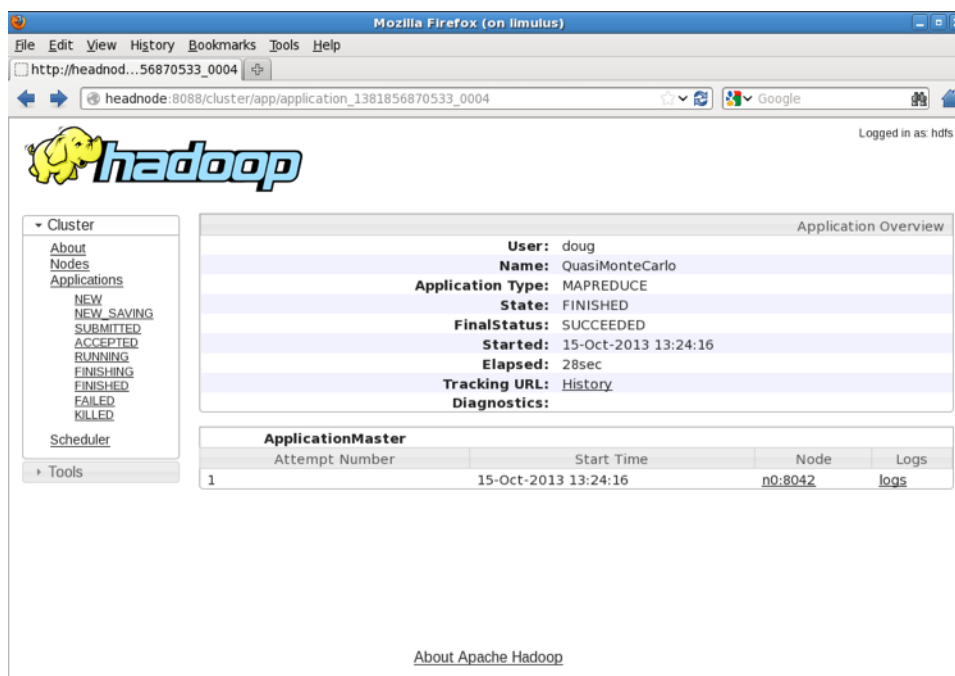
The screenshot shows a Mozilla Firefox browser window with the address bar displaying `n0:8042/node/containerlogs/container_1381856870533_0004_01_000001/doug`. The page title is "Logs for container_1381856870533_0004_01_000001". The Hadoop logo is visible in the top left. The page is logged in as "hdfs". A sidebar on the left contains links for "ResourceManager", "RM Home", "NodeManager", and "Tools". The main content area displays the following information:

Logs for container_1381856870533_0004_01_000001

stderr : Total file length is 2035 bytes.
stdout : Total file length is 0 bytes.
syslog : Total file length is 133148 bytes.

At the bottom of the page, there is a link for "About Apache Hadoop".

If you navigate back to the main Cluster page and select **Applications > Finished**, and then select the completed job, a summary page is displayed:



There are a few things to take note of based on our movement through the preceding GUI. First, because YARN manages applications, all input from YARN refers to an “application.” YARN has no data about the actual application. Data from the MapReduce job is provided by the MapReduce Framework. Thus there are two clearly different data streams that are combined in the web GUI: YARN applications and Framework jobs. If the Framework does not provide job information, certain parts of the web GUI will have nothing to display.

Another interesting aspect to note is the dynamic nature of the mapper and reducer tasks. These are executed as YARN Containers, and their numbers will change as the application runs. This feature provides much better cluster utilization, because mappers and reducers are dynamic rather than fixed resources.

Finally, there are other links in the preceding GUI that can be explored. With the MapReduce framework, it is possible to drill down to the individual map and reduce tasks. If log aggregation is enabled, the individual logs for each map and reduce task can be viewed.

Running the Terasort Test

To run the terasort benchmark, three separate steps are required. In general the rows are 100 bytes long, thus the total amount of data written is 100 times the number of rows (i.e. to write 100 GB of data, use 1000000000 rows). You will also need to specify input and output directories in HDFS.

1. Run `teragen` to generate rows of random data to sort.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.1.0-beta.jar teragen
<number of 100-byte rows> <output dir>
```

2. Run `terasort` to sort the database.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.1.0-beta.jar terasort
<input dir> <output dir>
```

3. Run teravalidate to validate the sorted Teragen.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-examples-2.1.0-beta.jar
teravalidate <terasort output dir> <teravalidate output dir>
```

Run the TestDFSIO Benchmark

YARN also includes a HDFS benchmark application named TestDFSIO. As with terasort, it requires several steps. Here we will write and read ten 1 GB files.

1. Run TestDFSIO in write mode and create data.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-client-jobclient-2.1.0-beta-tests.
jar TestDFSIO -write -nrFiles 10 -fileSize 1000
```

Example results are as follows (date and time removed):

```
fs.TestDFSIO: ----- TestDFSIO ----- : write
fs.TestDFSIO:           Date & time: Wed Oct 16 10:58:20 EDT 2013
fs.TestDFSIO:           Number of files: 10
fs.TestDFSIO: Total MBytes processed: 10000.0
fs.TestDFSIO:           Throughput mb/sec: 10.124306231915458
fs.TestDFSIO: Average IO rate mb/sec: 10.125661849975586
fs.TestDFSIO: IO rate std deviation: 0.11729341192174683
fs.TestDFSIO:           Test exec time sec: 120.45
fs.TestDFSIO:
```

2. Run TestDFSIO in read mode.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-client-jobclient-2.1.0-beta-tests.
jar TestDFSIO -read -nrFiles 10 -fileSize 1000
```

Example results are as follows (date and time removed):

```
fs.TestDFSIO: ----- TestDFSIO ----- : read
fs.TestDFSIO:           Date & time: Wed Oct 16 11:09:00 EDT 2013
fs.TestDFSIO:           Number of files: 10
fs.TestDFSIO: Total MBytes processed: 10000.0
fs.TestDFSIO:           Throughput mb/sec: 40.946519750553804
fs.TestDFSIO: Average IO rate mb/sec: 45.240928649902344
fs.TestDFSIO: IO rate std deviation: 18.27387874605978
fs.TestDFSIO:           Test exec time sec: 47.937
fs.TestDFSIO:
```

3. Clean up the TestDFSIO data.

```
yarn jar $YARN_EXAMPLES/hadoop-mapreduce-client-jobclient-2.1.0-beta-tests.
jar TestDFSIO -clean
```

1.3.1.2. MapReduce Compatibility

MapReduce was the original use case on which Hadoop was founded. In order to graph the World Wide Web and how it changes over time, MapReduce was developed to process this graph with billions of nodes and trillions of edges. Moving this technology to YARN made it a complex application to build due to requirements for data-locality, fault tolerance, and application priorities.

In order to provide data locality, the MapReduce Application Master is required to locate blocks for processing, and then request Containers on these blocks. To implement fault-tolerance, the ability to handle failed map or reduce tasks and request them again on other nodes was needed. Fault-tolerance moved hand-in-hand with the complex intra-application priorities.

The logic to handle complex intra-application priorities for map and reduce tasks had to be built as part of the Application Master. There was no need to start idle reducers before mappers finished processing enough data. Reducers were now under control of the Application Master and were not fixed as in Hadoop Version 1. One rather unique failure mode occurs when a node fails after all the maps have finished. When this happens, the map task must be repeated because the results are unavailable. In many cases all available Containers are being used by the reducer tasks, preventing the spawning of another mapper task to process the missing data. Logically this would create a deadlock with reducers waiting for missing mapper data. The MapReduce Application Master has been designed to detect this situation and, while not ideal, will kill enough reducers to free enough resources for mappers to finish processing the missing data. The killed reducers will start again, allowing the job to complete.

1.3.1.3. The MapReduce Application Master

The MapReduce Application Master is implemented as a composition of loosely coupled services. The services interact with each other via events. Each component acts on the received events and sends out any required events to other components. This design keeps it highly concurrent, with minimal or no synchronization required. The events are dispatched by a central Dispatch mechanism. All components register with the Dispatcher. The information is shared across different components using ApplicationContext.

In Hadoop 1 the death of the Job Tracker would result in the loss of all jobs – both running and queued. With a YARN MapReduce job, the equivalent to the Job Tracker is the Application Master. Because the Application Master will now run on compute nodes, this can lead to an increase in failure scenarios. To combat MapReduce Application Master failures, YARN has the capability to restart a specified number of times, as well as the capability to recover completed tasks. Additionally, much like the Job Tracker, the Application Master keeps metrics for jobs that are currently running. Typically the Application Master tracking URL makes these available, and these metrics can be found in the YARN web UI (See the previous pi example). The following settings can enable MapReduce recovery in YARN.

Enabling Application Master Restarts

To enable Application Master restarts:

1. You can adjust the `yarn.resourcemanager.am.max-retries` property in the `yarn-site.xml` file. The default setting is 2.
2. You can more directly tune how many times a MapReduce Application Master should restart by adjusting the `mapreduce.am.max-attempts` property in the `mapred-site.xml` file. The default setting is 2.

Enabling Recovery of Completed Tasks

You can use the `yarn.app.mapreduce.am.job.recovery.enable` property in the `yarn-site.xml` file to enable recovery of completed tasks. The default setting is "true".

The Job History Server

With the Application Master now taking the place of the Job Tracker, a centralized location to store the history of all MapReduce jobs was required. The Job History Server helps fill the void left by the transitory Application Master by hosting these completed job metrics and logs. This new history daemon is unrelated to the services provided by YARN, and is directly related to the MapReduce application framework.

1.3.1.4. Calculating the Capacity of a Node

Because YARN has now removed the hard partitioned mapper and reducer slots of Hadoop Version 1, new capacity calculations are required. There are eight important parameters for calculating a node's capacity that are specified in `mapred-site.xml` and `yarn-site.xml`:

In `mapred-site.xml`:

- `mapreduce.map.memory.mb`
`mapreduce.reduce.memory.mb`

These are the hard limits enforced by Hadoop on each mapper or reducer task.

- `mapreduce.map.java.opts`
`mapreduce.reduce.java.opts`

The heapsize of the `jvm -Xmx` for the mapper or reducer task. Remember to leave room for the JVM Perm Gen and Native Libs used. This value should always be lower than `mapreduce.[map|reduce].memory.mb`.

In `yarn-site.xml`:

- `yarn.scheduler.minimum-allocation-mb`

The smallest container that YARN will allow.

- `yarn.scheduler.maximum-allocation-mb`

The largest container that YARN will allow.

- `yarn.nodemanager.resource.memory-mb`

The amount of physical memory (RAM) for Containers on the compute node. It is important that this is not equal to the total amount of RAM on the node, as other Hadoop services also require RAM.

- `yarn.nodemanager.vmem-pmem-ratio`

The amount of virtual memory that each Container is allowed. This can be calculated with:

`containerMemoryRequest*vmem-pmem-ratio`

As an example, consider a configuration with the settings in the following table.

Example YARN MapReduce Settings

Property	Value
<code>mapreduce.map.memory.mb</code>	1536
<code>mapreduce.reduce.memory.mb</code>	2560
<code>mapreduce.map.java.opts</code>	<code>-Xmx1024m</code>
<code>mapreduce.reduce.java.opts</code>	<code>-Xmx2048m</code>
<code>yarn.scheduler.minimum-allocation-mb</code>	512
<code>yarn.scheduler.maximum-allocation-mb</code>	4096
<code>yarn.nodemanager.resource.memory-mb</code>	36864
<code>yarn.nodemanager.vmem-pmem-ratio</code>	2.1

With these settings, each map and reduce task has a generous 512MB of overhead for the Container, as evidenced by the difference between the `mapreduce.[map|reduce].memory.mb` and the `mapreduce.[map|reduce].java.opts`.

Next, YARN has been configured to allow a Container no smaller than 512MB and no larger than 4GB. The compute nodes have 36GB of RAM available for Containers. With a virtual memory ratio of 2.1 (the default value), each map can have up to 3225.6MB of RAM, or a reducer can have 5376MB of virtual RAM.

This means that the compute node configured for 36GB of Container space can support up to 24 maps or 14 reducers, or any combination of mappers and reducers allowed by the available resources on the node.

For more information about calculating memory settings, see [Determine YARN and MapReduce Memory Configuration Settings](#).

1.3.1.5. Changes to the Shuffle Service

As in Hadoop Version 1, the shuffle is required for parallel MapReduce job operation. Reducers fetch the outputs from all of the maps by “shuffling” map-output data from the corresponding nodes where map-tasks have run. The MapReduce Shuffle functionality is implemented as an Auxiliary Service in the Node Manager. This service starts up a Netty Web Server in the Node Manager address space, and knows how to handle MapReduce-specific shuffle requests from Reduce tasks. The MapReduce Application Master specifies the service ID for the shuffle service, along with security tokens that may be required when it starts any Container. In the returning response, the Node Manager provides the Application Master with the port on which the shuffle service is running, which is then passed on to the reduce tasks.

Hadoop Version 2 also provides the option for Encrypted Shuffle. With Encrypted Shuffle, the ability to use HTTPS with optional client authentication is provided. This feature is implemented with a toggle for HTTP or HTTPS, keystore/truststore properties, and the distribution of these stores to new or existing nodes. For details about the multi-step configuration of Encrypted Shuffle, review the Encrypted Shuffle documentation on the Apache Hadoop website.

1.3.1.6. Running Existing Hadoop Version 1 Applications on YARN

To ease the transition from Hadoop Version 1 to YARN, a major goal of YARN and the MapReduce framework implementation on top of YARN was to ensure that existing

MapReduce applications that were programmed and compiled against previous MapReduce APIs (we'll call these MRv1 applications) can continue to run with little or no modification on YARN (we'll refer to these as MRv2 applications).

Binary Compatibility of `org.apache.hadoop.mapred` APIs

For the vast majority of users who use the `org.apache.hadoop.mapred` APIs, MapReduce on YARN ensures full binary compatibility. These existing applications can run on YARN directly without recompilation. You can use `.jar` files from your existing application that code against `mapred` APIs, and use `bin/hadoop` to submit them directly to YARN.

Source Compatibility of `org.apache.hadoop.mapreduce` APIs

Unfortunately, it was difficult to ensure full binary compatibility to the existing applications that compiled against MRv1 `org.apache.hadoop.mapreduce` APIs. These APIs have gone through many changes. For example, several classes stopped being abstract classes and changed to interfaces. Therefore, the YARN community compromised by only supporting source compatibility for `org.apache.hadoop.mapreduce` APIs. Existing applications that use MapReduce APIs are source-compatible and can run on YARN either with no changes, with simple recompilation against MRv2 `.jar` files that are shipped with Hadoop 2, or with minor updates.

Compatibility of Command-line Scripts

Most of the command line scripts from Hadoop 1.x should simply just work. The only exception is MRAdmin, which was removed from MRv2 because JobTracker and TaskTracker no longer exist. The MRAdmin functionality is now replaced with RMAdmin. The suggested method to invoke MRAdmin (also RMAdmin) is through the command line, even though one can directly invoke the APIs. In YARN, when MRAdmin commands are executed, warning messages will appear reminding users to use YARN commands (i.e., RMAdmin commands). On the other hand, if applications programmatically invoke MRAdmin, they will break when running on YARN. There is no support for either binary or source compatibility.

Compatibility Trade-off Between MRv1 and Early MRv2 (0.23.x) Applications

Unfortunately, there are some APIs that are compatible with either MRv1 applications, or with early MRv2 applications (in particular, the applications compiled against Hadoop 0.23), but not both. Some of the APIs were exactly the same in both MRv1 and MRv2 except for the return type change in method signatures. Therefore, it was necessary to make compatibility trade-offs between the two.

- The `mapred` APIs are compatible with MRv1 applications, which have a larger user base.
- If `mapreduce` APIs didn't significantly break Hadoop 0.23 applications, they were made to be compatible with 0.23, but only source compatible with 1.x.

The following table lists the APIs that are incompatible with Hadoop 0.23. If early Hadoop 2 adopters using 0.23.x used the following methods in their custom routines, they will need to modify the code accordingly. For some problematic methods, an alternative method is provided with the same functionality and a method signature similar to MRv2 applications.

MRv2 Incompatible APIs

Problematic Method – org.apache.hadoop

util.ProgramDriver#drive

mapred.jobcontrol.Job#getMapredJobID

mapred.TaskReport#getTaskId

mapred.ClusterStatus #UNINITIALIZED_MEMORY_VALUE

mapreduce.filecache.DistributedCache #getArchiveTimestamps

mapreduce.filecache.DistributedCache #getFileTimestamps

mapreduce.Job#failTask

mapreduce.Job#killTask

mapreduce.Job#getTaskCompletionEvents

Incompatible Return Type Change

void -> int

String -> JobID

String -> TaskID

long -> int

long[] -> String[]

long[] -> String[]

void -> boolean

void -> boolean

mapred.TaskCompletionEvent[] -> mapreduce.TaskCompletionEvent[]

1.3.1.7. Running Existing MapReduce Version 1 Code on YARN

Running the Examples

Most of the MRv1 examples continue to work on YARN, but they are now in a new version of the .jar file. One exception worth mentioning is that the `sleep` example that used to be in `hadoop-examples-1.x.x.jar` is not in `hadoop-mapreduce-examples-2.x.x.jar`, but was moved into the test file `hadoop-mapreduce-client-jobclient-2.x.x-tests.jar`.

That exception aside, users may want to try running `hadoop-examples-1.x.x.jar` directly on YARN. Running `hadoop-jar hadoop-examples-1.x.x.jar` will still pick the classes in `hadoop-mapreduce-examples-2.x.x.jar`. This behavior is due to Java first searching for the desired class in the system .jar files. If the class is not found, it will go on to search in the user .jar files in the `classpath`. `hadoop-mapreduce-examples-2.x.x.jar`, which is installed with other MRv2 .jar files in the Hadoop classpath. Thus the desired class (e.g., `WordCount`) will instead be picked from this 2.x.x .jar file. However, it is possible to let Java pick the classes from the .jar file which is specified after the `-jar` option. There are two options:

- Add `HADOOP_USER_CLASSPATH_FIRST=true` and `HADOOP_CLASSPATH=...:hadoop-examples-1.x.x.jar` as environment variables, and add `mapreduce.job.user.classpath.first = true` in `mapred-site.xml`.
- Remove the 2.x.x .jar file from the classpath. If it is a multiple-node cluster, the .jar file needs to be removed from the classpath on all the nodes.

Running Apache Pig Scripts on YARN

Apache Pig is one of the two major data processing applications in the Hadoop ecosystem, the other being Hive. Due to significant efforts from the Pig community, existing Pig scripts do not require any modifications. Pig on YARN in Hadoop 0.23 has been supported since Pig version 0.10.0, and Pig working with Hadoop 2.x has been supported starting with Pig version 0.10.1.

Existing Pig scripts that work with Pig version 0.10.1 and beyond will work just fine on YARN, however, versions earlier than Pig 0.10.x may not run directly on YARN due to some incompatible MapReduce APIs and configuration.

Running Apache Hive Queries on YARN

Existing Hive queries do not need any changes to work on YARN starting with Hive-0.10.0, thanks to the work done by Hive community. Support for Hive on YARN in Hadoop 0.23 and 2.x releases has been in place since Hive-0.10.0. Queries that work on Hive-0.10.0 and beyond will work without changes on YARN. However, as with Pig, earlier versions of Hive may not run directly on YARN, as those Hive releases do not support 0.23 and 2.x.

Running Apache Oozie Workflows on YARN

As with Pig and Hive, the Apache Oozie community worked to make sure existing Oozie workflows run in a completely backwardly-compatible manner. Support for Hadoop 0.23 and 2.x is available starting Oozie release 3.2.0. Existing Oozie workflows can start taking advantage of YARN in 0.23 and 2.x with Oozie 3.2.0 and above.

1.3.1.8. Uber Jobs (Technical Preview)



Note

This feature is a technical preview and considered under development. Do not use this feature in your production systems. If you have questions regarding this feature, contact Support by logging a case on our Hortonworks Support Portal at <https://support.hortonworks.com>.

An Uber Job is when multiple mappers and reducers are combined to use a single Container. There are four core settings around the configuration of Uber Jobs found in the `mapred-site.xml` options presented in the following table.

Configuration options for Uber Jobs

Property	Description
<code>mapreduce.job.ubertask.enable</code>	Whether to enable the small-jobs "ubertask" optimization, which runs "sufficiently small" jobs sequentially within a single JVM. "Small" is defined by the following <code>maxmaps</code> , <code>maxreduces</code> , and <code>maxbytes</code> settings. Users can override this value. Default = false
<code>mapreduce.job.ubertask.maxmaps</code>	The threshold for the number of maps beyond which a job is considered too large for the ubertasking optimization. Users can override this value, but only downward. Default = 9
<code>mapreduce.job.ubertask.maxreduces</code>	The threshold for the number of reduces beyond which a job is considered too large for the ubertasking optimization. CURRENTLY THE CODE CANNOT SUPPORT MORE THAN ONE REDUCE and will ignore larger values (zero is a valid maximum value, however). Users can override this value, but only downward.
<code>mapreduce.job.ubertask.maxbytes</code>	Default = 1 The threshold for the number of input bytes beyond which a job is considered too large for the ubertasking optimization. If no value is specified, <code>dfs.block.size</code> is used as the default. Be sure to specify a default value in <code>mapred-site.xml</code> if the underlying file system is not HDFS. Users can override this value, but only downward.

Default = HDFS Block Size

1.3.2. MapReduce Version 2 Troubleshooting Guide

This section presents basic information to assist with troubleshooting MapReduce Version 2 (MRv2) issues on the Hortonworks Data Platform (HDP). This document is not meant to be a comprehensive guide for all MRv2 issues. It provides basic techniques that can be used to isolate issues and help lead to a solution.

Use the following resources to learn more about Troubleshooting MapReduce Version 2:

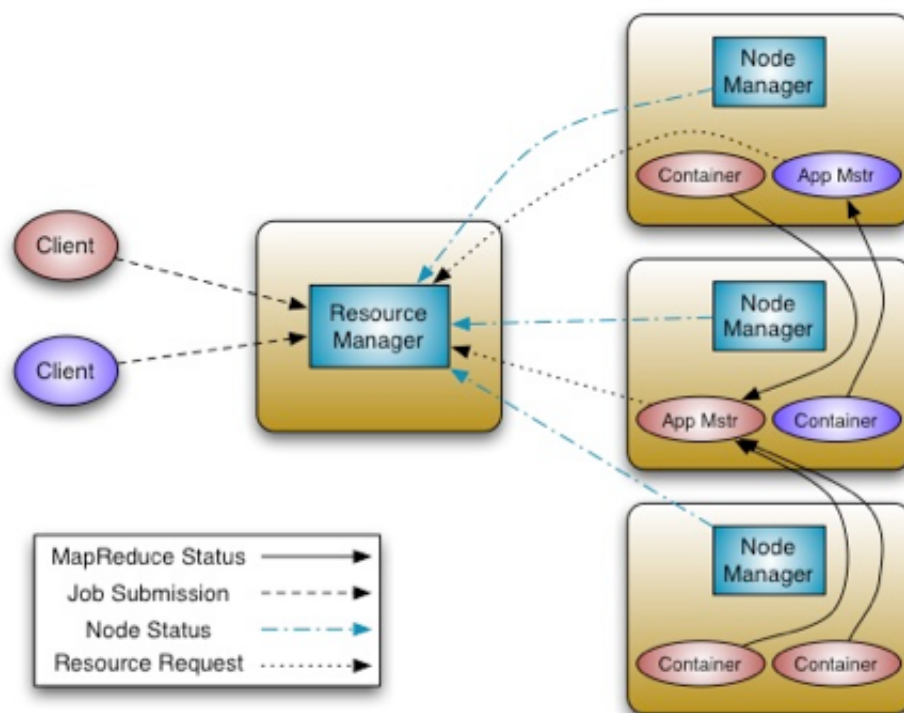
- [MRv2 Overview](#)
- [MRv2 Terminology](#)
- [Configuration Files](#)
- [Process ID Files](#)
- [Log Files](#)
- [Ports](#)
- [Gathering General Information](#)
- [Common MRv2 Commands](#)
- [MRv2 Troubleshooting Actions Checklist](#)
- [Common Server-Side Issues](#)
- [Common Client-Side Issues](#)

1.3.2.1. MRv2 Overview

In Hadoop version 1, MapReduce was responsible for both processing and cluster resource management. In Apache Hadoop version 2, cluster resource management has been moved from MapReduce into YARN, thus enabling other application engines to utilize YARN and Hadoop, while also improving the performance of MapReduce.

The fundamental idea of YARN is to split up the two major responsibilities of scheduling jobs and tasks that were handled by the version 1 MapReduce Job Tracker and Task Tracker into separate entities:

- Resource Manager – replaces the Job Tracker.
- Application Master – a new YARN component
- Node Manager – replaces the Task Tracker
- Container running on a Node Manager – replaces the MapReduce slots



Resource Manager

In YARN, the Resource Manager is primarily a pure scheduler. In essence, it is strictly limited to arbitrating available resources in the system among the competing applications. It optimizes for cluster utilization (keeps all resources in use all the time) against various constraints such as capacity guarantees, fairness, and SLAs. To allow for different policy constraints the Resource Manager has a pluggable scheduler that allows for different algorithms, such as capacity, to be used as necessary. The daemon runs as the "yarn" user.

Node Manager

The Node Manager is YARN's per-node agent, and takes care of the individual compute nodes in a Hadoop cluster. This includes keeping up-to-date with the Resource Manager, life-cycle management of application Containers, monitoring resource usage (memory, CPU) of individual Containers, monitoring node health, and managing logs and other auxiliary services that can be utilized by YARN applications. The Node Manager will launch Containers ranging from simple shell scripts to C, Java, or Python processes on Unix or Windows, or even full-fledged virtual machines (e.g. KVMs). The daemon runs as the "yarn" user.

Application Master

The Application Master is, in effect, an instance of a framework-specific library and is responsible for negotiating resources from the Resource Manager and working with the Node Manager(s) to execute and monitor the Containers and their resource consumption. It has the responsibility of negotiating appropriate resource Containers from the Resource Manager and monitoring their status. The Application Master runs as a single Container that is monitored by the Resource Manager.

Container

A Container is a resource allocation, which is the successful result of the Resource Manager granting a specific Resource Request. A Container grants rights to an application to use a specific amount of resources (memory, CPU, etc.) on a specific host. In order to utilize Container resources, the Application Master must take the Container and present it to the Node Manager managing the host on which the Container was allocated. The Container allocation is verified in secure mode to ensure that Application Master(s) cannot fake allocations in the cluster.

Job History Service

This is a daemon that serves historical information about completed jobs. We recommend running it as a separate daemon. Running this daemon consumes considerable HDFS space as it saves job history information. This daemon runs as the “mapred” user.

1.3.2.2. MRv2 Terminology

Resource Model

YARN supports a very general resource model for applications. An application (via the Application Master) can request resources with highly specific requirements such as:

- Resource name (more complex network topologies that would include host name and rack name are currently under development)
- Memory (in MB)
- CPU (cores, for now)

Container Specification During Launch:

While a Container is merely a right to use a specified amount of resources on a specific machine (Node Manager) in the cluster, the Application Master has to provide considerably more information to the Node Manager to actually launch the Container. This information can include .jar files, memory requested, input data, and number of CPUs.

Resource Request

YARN is designed to allow individual applications (via the Application Master) to utilize cluster resources in a shared, secure, and multi-tenant manner. YARN also remains aware of cluster topology in order to efficiently schedule and optimize data access (i.e. reduce data motion) for applications to the greatest possible extent.

In order to meet those goals, the central Scheduler (in the Resource Manager) has extensive information about an application’s resource needs, which allows it to make better scheduling decisions across all applications in the cluster. This leads to the Resource Request and the resulting Container.

Resource-requirement

These are the resources required to start a Container, such as memory, CPU, etc.

Number-of-containers

This is essentially the unit of MRv2 for each job. Keep in mind that the Application Master runs as a Container. Up to 10% of the capacity can be allocated to the Application Master Container.

Map Phase

The first phase of a MRv2 job which runs locally with respect to the data. It takes the input data, processes it into key-value pairs, and outputs it to the reduce phase. The data is passed into the Mapper as a <key, value> pair generated by an InputFormat instance. InputFormat determines where the input data needs to be split between the Mappers, and then generates an InputSplit instance for each split. The Partitioner creates the partition for the record. This determines which reducer will process the record.

After processing this data, the map method of the Mapper class outputs a <key, value> pair that is stored in an unsorted buffer in memory. When the buffer fills up, or when the map task is complete, the <key, value> pairs in the buffer are sorted, then spilled to the disk. If more than one spill file was created, these files are merged into a single file of sorted <key, value> pairs. The sorted records in the spill file wait to be retrieved by a Reducer.

Mapper

The individual task of the map phase as run by the Containers. MRv2 spawns a map task for each InputSplit generated by the InputFormat.

Reduce Phase

The final phase of a MRv2 job, the Reduce phase gathers and combines the output of all of the mappers. The Reducer fetches the records from the Mapper. The Reduce phase can be broken down into three phases:

- **Shuffle:** This occurs when MapReduce retrieves the output of the Mappers (via HTTP) and sends the data to the Reducers. All records with the same key are combined and sent to the same Reducer.
- **Sort:** This phase happens simultaneously with the shuffle phase. As the records are fetched and merged, they are sorted by key.
- **Reduce:** The reduce method is invoked for each key.

Reducer

The individual task of the Reduce phase run by the Containers after the map phase has completed for at least one of the mappers.

Task

The partition of a job that will be run on a Container, which keeps track of the status of the MRv2 tasks that have been assigned to it.

Container Executor

Controls initialization, finalization, clean up, launching, and killing of the task's JVM. This is the equivalent of the Task Tracker controller in MRv1.

The Container Executor is set up in two files:

container-executor.cfg – a file which has the following configuration by default:

- Conf name `yarn.nodemanager.linux-container-executor.group` has a value of "hadoop"
- Conf name `banned.users` has a value of "hdfs,yarn,mapred,bin"
- Conf name `min.user.id` has a value of "1000"

yarn-site.xml – a file with the following configuration:

- Conf name `yarn.nodemanager.container-executor.class` has a value of "org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor"
- Conf name `yarn.nodemanager.linux-container-executor.group` has a value of "hadoop"

Capacity Scheduler

The Capacity Scheduler is designed to run Hadoop MRV2 as a shared, multi-tenant cluster in an operator-friendly manner, while also maximizing the throughput and utilization of the cluster when running MRv2 applications.

Users submit jobs to Queues. Queues, as a collection of jobs, allow the system to provide specific functionality. HDP comes configured with a single mandatory queue named "default".

For more information about the Scheduler, see the [Capacity Scheduler](#) guide.

ACL

The Access Control List (ACL) can be enabled on the cluster for job-level and queue-level authorization. When enabled, access control checks are performed by:

- The Resource Manager before allowing users to submit jobs to queues and administering these jobs.
- The Resource Manager and the Node Manager before allowing users to view job details or to modify a job using MRv2 APIs, CLI, or Web user interfaces.

There is no need for the `mapred-queue-acl.xml` since it is all configured in the `capacity-scheduler.xml` file.

Data Compression

MapReduceV2 provides facilities for the application writer to specify compression for both intermediate map-outputs and the job-outputs (i.e., the output of the reducers). It can be set up with CompressionCodec implementation for the zlib compression algorithm. The gzip file format is also supported.

Concatenation

MapReduceV2 can concatenate multiple small files into one block size that is more efficient in storage and data movement.

Distributed Cache

DistributedCache efficiently distributes large application-specific, read-only files. The framework will copy the necessary files to the slave node before any tasks for the job are executed on that node. It is also very common to use the DistributedCache by using the GenericOptionsParser. For example:

```
$ bin/hadoop jar -libjars testlib.jar -files file.txt args
```

The above command will copy the file.txt file to the cluster.

Hadoop Pipes

Hadoop Pipes is the name of the C++ interface to Hadoop MRv2. Hadoop Pipes use sockets as the channel over which the Node Manager communicates with the process running the C++ map or reduce function. JNI is not used.

Hadoop Streaming

This is a Hadoop API to MRv2 that allows the user to write map and reduce functions in languages other than Java (Perl, Python, etc.). Hadoop Streaming uses Windows streams as the interface between Hadoop and the program, so the user can use any language that can read standard input and write to standard output to write the MapReduce program. Streaming is naturally suited for text processing.

Security

MRv2 supports Kerberos security and can run on Kerberos-secured clusters. Extra configuration may be required to achieve functioning of MRv2 on a secured cluster.

Job .jar Files

MRv2 sets a value for the replication level for submitted job .jar files. As a best practice, set this value to approximately the square root of the number of nodes. The default value is 10.

Note: The path for hadoop .jar files is different:

```
/usr/lib/hadoop-mapreduce/
```

1.3.2.3. Configuration Files

These files are used to configure MapReduce jobs.



Note

Default paths of the files are provided as they are from an HDP install. Users may choose to change these locations as per need.

- `/etc/hadoop/conf/yarn-site.xml`

This file contains configuration settings for YARN. It is used by the Client, the Node Manager, and the Resource Manager. The following table lists some important `yarn-site.xml` properties.

Property	Value
yarn.resourcemanager.webapp.address	<RM_HOST>:8088
yarn.log.server.url	<H_S>:19888/jobhistory/logs
yarn.resourcemanager.hostname	<RM_HOST>
yarn.nodemanager.linux-container-executor.group	hadoop
yarn.nodemanager.log.retain-second	604800
yarn.log-aggregation-enable	true
yarn.nodemanager.container-monitor.interval-ms	3000
yarn.nodemanager.log-aggregation.compression-type	gz

- **/etc/hadoop/conf/core-site.xml**

This file contains configuration settings for Hadoop Core, such as I/O settings that are common to HDFS2 and MRv2. It is used by all Hadoop daemons and clients, because all daemons need to know the location of the Name Node. Hence this file should have a copy in each node running a Hadoop daemon or client.

- **/etc/hadoop/conf/mapred-site.xml**

This file contains configuration settings for MRv2 properties such as `io.sort` and memory settings for the Containers. The following table lists some important mapred-site.xml properties.

Property	Value	De
mapreduce.map.memory.mb	1024	#1
mapreduce.reduce.memory.mb	1024	#2
mapreduce.map.java.opts	-Xmx756m	Th
mapreduce.reduce.java.opts	-Xmx756m	Th
mapreduce.reduce.log.level	INFO	log
mapreduce.jobhistory.done-dir	/mr-history/done	Th
mapreduce.shuffle.port	13562	En
yarn.app.mapreduce.am.staging-dir	/user	Th
mapreduce.reduce.shuffle.parallelcopies	30	Sc
mapreduce.framework.name	yarn	Bar

- **/etc/hadoop/conf/capacity-scheduler.xml**

This is the configuration file for the Capacity Scheduler component in the Hadoop Resource Manager. You can use this file to configure various scheduling parameters related to queues.

- **/etc/hadoop/conf/hadoop-env.sh**

Java is required by Hadoop, so this file is used by the HDFS daemons to locate `JAVA_HOME`. This file also specifies memory settings for all of the HDFS daemons. This is the file to use if you need to tweak memory settings for the HDFS daemons. This file might also be investigated when dealing with memory errors with the HDFS daemons. This file is useful for memory issues and garbage collector issues.

- **/etc/hadoop/conf/yarn-env.sh**

Java is required by Hadoop, so this file is used by the YARN daemons to locate JAVA_HOME. This file also specifies memory settings for all of the YARN daemons. This is the file to use if you need to tweak memory settings for the YARN daemons. This file might also be investigated when dealing with memory errors with the YARN daemons. This file is also useful for memory issues and garbage collector issues.

- **/etc/hadoop/conf/log4j.properties**

This file is used to modify the log purging intervals of the MapReduce log files. It defines the logging for all of the Hadoop daemons, and includes information related to appenders used for logging and layout.

Configuration File Permissions

Listed below are the proper HDFS-related permissions and user/groups for folders and files for a working HDP cluster.

```
drwxr-xr-x 3 root root 4096 /etc/hadoop
lrwxrwxrwx 1 hadoop_deploy hadoop 29 conf -> /etc/alternatives/hadoop-conf
-rw-r--r-- 1 hdfs hadoop 2316 core-site.xml
-rw-r--r-- 1 mapred hadoop 7632 mapred-site.xml
-rw-r--r-- 1 mapred hadoop 7632 yarn-site.xml
-rw-r--r-- 1 mapred hadoop 2033 mapred-queue-acls.xml
-rw-r--r-- 1 hdfs hadoop 928 taskcontroller.cfg
-rw-r--r-- 1 root root 9406 capacity-scheduler.xml
-rw-r--r-- 1 root root 327 fair-scheduler.xml
-rw-r--r-- 1 hdfs hadoop 4867 hadoop-env.sh
-rw-r--r-- 1 hdfs hadoop 4867 yarn-env.sh
```

1.3.2.4. Process ID Files

These are the process ID (.pid) files that are generated when the daemons are created:

```
/var/run/hadoop-yarn/yarn/yarn-yarn-resourcemanager.pid
/var/run/hadoop-yarn/yarn/yarn-yarn-nodemanager.pid
/var/run/hadoop-mapreduce/mapred/mapred-mapred-historyserver.pid
```

Without the proper permissions, the MapReduce daemons may not start or stop. The proper permissions are listed below.

```
drwxr-xr-x 2 yarn hadoop 4096 Oct 31 11:54 .
drwxrwxr-x 3 yarn hadoop 4096 Oct 20 15:15 ..
-rw-r--r-- 1 yarn hadoop 5 Oct 31 11:54 yarn-yarn-nodemanager.pid
-rw-r--r-- 1 yarn hadoop 5 Oct 31 11:54 yarn-yarn-resourcemanager.pid
-rw-r--r-- 1 mapred hadoop 5 Oct 31 11:54 /var/run/hadoop-mapreduce/mapred/
mapred-mapred-historyserver.pid
```

1.3.2.5. Log Files

These files are used to configure MapReduce jobs.

The following environment files define the log location for YARN and MRv2 for the daemons.

- yarn-env.sh: export YARN_LOG_DIR=/var/log/hadoop-yarn/\$USER

- `hadoop-env.sh:export HADOOP_LOG_DIR=/var/log/hadoop-mapred/$USER`



Note

- The Job History Server runs as the “mapred” user, and it will use the `hadoop-env.sh` file.
- The Resource Manager and the Node Manager run as the “yarn” user, and they will use the `yarn-env.sh` file.
- The Application Master and Container log files are removed from the local directories once the Job finishes, and are moved to the HDFS2 directory.
- The Job History Server will display the Application Master and Containers log files, which are stored in HDFS2 for 30 days.

Daemon .out Files

The log files with the `.out` extension for MRv2 and YARN daemons are located in `/var/log/hadoop-mapred/mapred` and in `/var/log/hadoop-yarn/yarn`. These files have the following naming convention:

- `hadoop-mapred-historyserver-<HistoryServer_Host>.out`
- `yarn-yarn-nodemanager-<nodemanager_host>.out`
- `yarn-yarn-resourcemanager-<resourcemanager_host>.out`

These `.out` files are created and written to during start-up of the MRv2 and YARN daemons. It is very rare that these files get populated, but they can be helpful when trying to determine why Resource Manager, Node Manager, or the Job History Server daemons are not starting up.

Daemon .log Files

The log files with the `.log` extension for MRv2 and YARN daemons are located in `/var/log/hadoop/mapred` and in `/var/log/hadoop-yarn/yarn`. These files have the following naming convention:

- `hadoop-mapred-historyserver-<HistoryServer_Host>.log`
- `yarn-yarn-nodemanager-<nodemanager_host>.log`
- `yarn-yarn-resourcemanager-<resourcemanager_host>.log`

These files show the log messages for the running daemons. If there are any errors encountered while the daemon is running, the stack trace of the error is logged in these files.



Note

Log files are rotated daily by default, but can be adjusted by modifying the `/etc/hadoop/conf/log4j.properties` file.

Daemon .log.<date> Files

The .log.<date>files have the following format:

- `hadoop-mapred-historyserver-<HistoryServer_Host>.log.<date>`
- `yarn-yarn-nodemanager-<nodemanager_host>.log.<date>`
- `yarn-yarn-resourcemanager-<resourcemanager_host>.log.<date>`

When .log files are rotated, the file name is appended with a date. An example of the file name would be:

```
mapred-mapred-historyserver-sandbox.log.2013-10-26
```

This indicates that the file was rotated on Oct 26, 2013. These files are useful when an issue has occurred multiple times, and comparing these older log files with the most recent log file can help uncover patterns of occurrence.

MapReduce V2 Container Log Files

The log files for specific applications and their Containers are located in HDFS. You can access Container log files using either the command line or the Resource Manager UI.

To access the Container log files using the command line, you first need to obtain the application ID by running the following command:

```
yarn application -list
```

This command returns a list of the applications, along with their application IDs:

```
yarn application -list
13/11/04 23:39:09 INFO client.RMProxy: Connecting to Resource Manager at
sandbox/10.11.2.159:8050
Total number of applications (application-types: [] and states: [SUBMITTED,
ACCEPTED, RUNNING]):1
```

User	Queue	Application-Id	Application-Name	Application-Type
		State	Final-State	Progress
		Tracking-URL		
application_1383601692319_0008		QuasiMonteCarlo		MAPREDUCE
hdfs	default	ACCEPTED	UNDEFINED	0%
		N/A		

You can then use the application ID in the following command to access the Container log files:

```
yarn logs -applicationId application_1383601692319_0008
```



Note

The log files are stored in HDFS under the following path:

```
/app-logs/hdfs/logs/<application_id>/<hostnames*>_4545
```

Information about using the Resource Manager UI to access the Container log files is available on [this page](#) under "Using the Web GUI to Monitor Examples."

1.3.2.6. Ports

This section provides information about the ports used by the MapReduce services. Because this is a distributed file system, this port information can help isolate issues due to communication between nodes.

HTTP Ports

The Resource Manager, Job History Server, and Node Manager services each have a web interface, and therefore listen on an Hypertext Transfer Protocol (HTTP) port. This makes it possible for any client with network access to the nodes to access the web page and view specific information on the MRv2 jobs submitted. Each of the MapReduce services can be configured to listen on a specific port. These ports are configured within the `yarn-site.xml` file. The following table lists the HTTP ports for the MRv2 and YARN services.

MapReduce Service	Configuration Property Name from <code>yarn-site.xml</code>
Resource Manager	<code>yarn.resourcemanager.webapp.address</code>
Job History Server	<code>yarn.log.server.url</code>

IPC Ports

Interprocess Communication (IPC) is used by the Resource Manager to submit jobs. The following table lists the ports that the Resource Manager uses to for job submission.

MapReduce service	Configuration Property Name from <code>yarn-site.xml</code>
Resource Manager	<code>yarn.resourcemanager.address</code>
RM APIs webapp	<code>yarn.resourcemanager.webapp.address</code>
RM scheduler	<code>yarn.resourcemanager.scheduler.address</code>
Node Manager	<code>yarn.nodemanager.address</code>
RM Admin	<code>yarn.resourcemanager.admin.address</code>

For more information on IPC, see <http://wiki.apache.org/hadoop/ipc>.

1.3.2.7. Gathering General Information

Operating System Information

The following commands will provide the Linux Kernel version and type. With this information, you can determine if HDP is running on a supported platform.

Command:

```
uname -a
```

Example:

```
$ uname -a
Linux test63.localdomain 2.6.32-279.el6.x86_64 #1 SMP Fri Jun 22
12:19:21 UTC 2012 x86_64 x86_64 x86_64 GNU/Linux
```

Command:

```
cat /proc/version
```

Example:

```
$ cat /proc/version
Linux version 2.6.32-279.el6.x86_64
(mockbuild@c6b9.bsys.dev.centos.org) (gcc version 4.4.6 20120305 (Red
Hat 4.4.6-4) (GCC) ) #1 SMP Fri Jun 22 12:19:21 UTC 2012
```

Command:

```
cat /etc/*-release
```

Example:

```
$ cat /etc/*-release
CentOS release 6.3 (Final)
CentOS release 6.3 (Final)
CentOS release 6.3 (Final)
```

Determine Installed Software

This information is helpful when troubleshooting performance-related issues, or when there is unexpected behavior occurring on one specific machine. One example would be a MapReduce job that suddenly starts running for 20 minutes rather than the expected 1 minute. The following command does not list any tarball-type installations, so you should keep in mind the possibility that some programs may have been installed outside of the system package manager.

Command:

```
rpm -qa
```

Example:

```
# rpm -qa | egrep "hadoop|yarn"
hadoop-hdfs-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-mapreduce-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-lzo-native-0.5.0-1.x86_64
hadoop-mapreduce-historyserver-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-lzo-0.5.0-1.x86_64
hadoop-yarn-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-libhdfs-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-yarn-resourcemanager-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-client-2.2.0.2.0.6.0-76.el6.x86_64
hadoop-yarn-nodemanager-2.2.0.2.0.6.0-76.el6.x86_64
```

Detect Running Processes

This information is helpful when troubleshooting performance-related issues, or when there is unexpected behavior occurring on one specific machine.

Command:

```
ps -aux
```


Example:

```
$ ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0  19348   620 ?        Ss   Sep25    0:06 /sbin/init
postgres  6705  0.0  0.0 214952  2936 ?        Ss   09:18    0:00 postgres:
   mapred ambarirca 10.10.3.27(60031) idle
root         3  0.0  0.0      0     0 ?        S    Sep25    0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S    Sep25    0:07 [ksoftirqd/0]
```

Detect Java Running Processes

The command below lists the Java processes that are running on the machine. Since most Hadoop code is based on Java, using this command can also help verify that the Hadoop processes are running on a specific machine.



Note

The `jps` command may not be within the `PATH` variable for the user that is logged in. If not, you must first set the `PATH` variable, or you can simply provide the full path, for example:

```
/usr/jdk64/jdk1.6.0_31/bin/jps
```

Command:

```
jps
```

Example output:

```
10528 Resource Manager
25185 Jps
9202 RunJar
10141 Bootstrap
8001 QuorumPeerMain
7357 NameNode
8358 HMaster
12474 HRegionServer
9605 RunJar
1921 Node Manager
8857 JobHistoryServer
5612 DataNode
17667 RunJar
2943 AmbariServer
11103 SecondaryNameNode
```

Show Open Files Linked to a Process ID

This information is helpful in determining which process has a lock on a specific file, such as issues where errors state that a file is locked, and hence a process cannot start because it cannot write to a file.

Command:

```
lsuf -p <pid> | grep <file string name>
```

Example:

```
$ lsof -p 8857 | grep var
java      8857 mapred    1w  REG                253,0      2031 542470 /var/log/
hadoop-mapreduce/mapred/mapred-mapred-historyserver-sandbox.out
java      8857 mapred    2w  REG                253,0      2031 542470 /var/log/
hadoop-mapreduce/mapred/mapred-mapred-historyserver-sandbox.out
java      8857 mapred   159w REG                253,0     95452 542286 /var/log/
hadoop-mapreduce/mapred/mapred-mapred-historyserver-sandbox.log
```

Verifying Well-formed XML

The following command can help determine if a configuration file in XML format is well-formed. If the XML file is well-formed, it will simply be opened. If there are problems with the file, a list of errors will be displayed. This command can help uncover any syntax errors that might occur as a result of manually editing configuration files in Hadoop. The example below shows the list of errors returned when an XML file is not well-formed.

Command:

```
xmllint <xml file>
```

Example:

```
$ xmllint ./hdfs-site.xml
./hdfs-site.xml:187: parser error : Opening and ending tag mismatch: property
line 6 and configuration
</configuration>
      ^
./hdfs-site.xml:188: parser error : Premature end of data in tag property line
3
^
./hdfs-site.xml:188: parser error : Premature end of data in tag configuration
line 2
^
```

Detect Auto-start Processes

Ideally the cluster administrator should have this information. Information about auto-start processes can help in determining why a certain behavior is specific to a machine. For instance, it is possible that a process that auto-starts on boot-up is preventing one of the HDP components to launch due to a port conflict after rebooting a node. Below is the command that returns a list of cron jobs.

Command:

```
for user in $(cut -f1 -d: /etc/passwd); do crontab -u $user -l; done
```

Example:

```
$ for user in $(cut -f1 -d: /etc/passwd); do crontab -u $user -l; done
no crontab for root
no crontab for bin
no crontab for daemon
no crontab for adm
no crontab for lp
no crontab for sync
```

```
no crontab for shutdown
no crontab for halt
no crontab for mail
```

Get a List of All Mounts on a Machine

Gathering this information will help in determining what drives are actually mounted or available for use on the node. The following commands are helpful in determining if the system is hitting any storage limitations. Unexpected behavior can occur when disks are full.

Command:

```
df
```

Example:

```
$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
/dev/mapper/VolGroup-lv_root
11272464      4729432   5970412   45% /
tmpfs                961928         272    961656    1% /dev/shm
/dev/sda1            495844      37433    432811    8% /boot
```

Command:

```
cat /etc/fstab
```

Example:

```
root@a2nn:~> cat /etc/fstab

#
# /etc/fstab
# Created by anaconda on Wed Mar 20 15:03:22 2013
#
# Accessible filesystems, by reference, are maintained under '/dev/disk'
# See man pages fstab(5), findfs(8), mount(8) and/or blkid(8) for more info
#
/dev/mapper/vg_a2nn-lv_root / ext4 defaults 1
1
UUID=8bbdbae7-9cb8-4b66-af1c-4f904f047501 /boot ext4
defaults 1 2
/dev/mapper/vg_a2nn-lv_swap swap swap defaults 0
0
tmpfs /dev/shm tmpfs defaults 0 0
devpts /dev/pts devpts gid=5,mode=620 0 0
sysfs /sys sysfs defaults 0 0
proc /proc proc defaults 0 0
root@a2nn:~>
```

Operating System Log Files

These files can help in determining if a machine was rebooted or shut down at a particular time. The log files can help determine why some HDP services were not working or not operational at a specific time.

- /var/log/messages

Contains global system messages, including messages that are logged during system start-up.

- `/var/log/audit/audit.log`

This file is only available if the `/etc/init.d/auditd` daemon has been started. To check status, execute `/etc/init.d/auditd status`. This file can be used to check which user executed a command at a particular time.

Hardware Information

The following commands provide information about the hardware components installed on the machine. This will help in isolating issues related to hardware.

The following command lists information about the PCI buses and devices in the system:

```
lspci
```

Example:

```
$ lspci
00:00.0 Host bridge: Intel Corporation 440FX - 82441FX PMC [Natoma] (rev 02)
00:01.0 ISA bridge: Intel Corporation 82371SB PIIX3 ISA [Natoma/Triton II]
00:01.1 IDE interface: Intel Corporation 82371AB/EB/MB PIIX4 IDE (rev 01)
00:02.0 VGA compatible controller: InnoTek Systemberatung GmbH VirtualBox
Graphics Adapter
00:03.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32
LANCE] (rev 10)
00:04.0 System peripheral: InnoTek Systemberatung GmbH VirtualBox Guest
Service
00:05.0 Multimedia audio controller: Intel Corporation 82801AA AC'97 Audio
Controller (rev 01)
00:06.0 USB controller: Apple Computer Inc. KeyLargo/Intrepid USB
00:07.0 Bridge: Intel Corporation 82371AB/EB/MB PIIX4 ACPI (rev 08)
00:08.0 Ethernet controller: Advanced Micro Devices [AMD] 79c970 [PCnet32
LANCE] (rev 10)
00:0d.0 SATA controller: Intel Corporation 82801HM/HEM (ICH8M/ICH8M-E) SATA
Controller [AHCI mode] (rev 02)
```

The following command returns CPU information for the machine:

```
cat /proc/cpuinfo
```

Example:

```
$ cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 58
model name : Intel(R) Core(TM) i7-3615QM CPU @ 2.30GHz
stepping : 9
cpu MHz : 2283.256
cache size : 6144 KB
```

Network Information

This set of commands can be helpful when troubleshooting network issues.

The following command provides the IP address and validates that the network interfaces are up, and that there is an IP address tied to the interfaces. When dealing with issues involving communication between nodes, this is a good place to start.

```
ifconfig
```

Example:

```
$ ifconfig
eth0      Link encap:Ethernet  HWaddr 08:00:27:76:CD:33
          inet addr:10.10.3.27  Bcast:10.10.3.255  Mask:255.255.254.0
          inet6 addr: fe80::a00:27ff:fe76:cd33/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1217765 errors:0 dropped:0 overruns:0 frame:0
          TX packets:336245 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:308949876 (294.6 MiB)  TX bytes:128725650 (122.7 MiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:1609854 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1609854 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:619945138 (591.2 MiB)  TX bytes:619945138 (591.2 MiB)

virbr0    Link encap:Ethernet  HWaddr 52:54:00:EB:5E:B7
          inet addr:192.168.122.1  Bcast:192.168.122.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:175 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 b)  TX bytes:10172 (9.9 KiB)
```

The following command returns a list of the ports used within the system. This is helpful in determining if a specific port is already in use, and therefore another application cannot bind and listen on the port. This command can be useful when trying to isolate why certain HDP Master processes are not able to start up.

```
netstat -an
```

Example:

```
$ netstat -an
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.10.10.157:51111      0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:199           0.0.0.0:*               LISTEN
tcp        0      0 10.10.10.157:50090      0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:8010            0.0.0.0:*               LISTEN
```

```
tcp      0      0 0.0.0.0:3306          0.0.0.0:*
LISTEN
tcp      0      0 0.0.0.0:8651          0.0.0.0:*
LISTEN
```

You can use the following command to start, stop, and check the status of the firewall service on a CentOS or RHEL operating system. The firewall service status represents another possible reason why communication cannot be established between nodes.

```
service iptables [stop | status | start]
```

Example:

```
$service iptables status
iptables: Firewall is not running.
```

1.3.2.8. Common MRv2 Commands

This section provides some helpful MapReduce Version 2 commands. The full list of commands is available on the apache.org website at <http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-common/CommandsManual.html>.

Getting the Hadoop Version

Command:

```
hadoop version
```

Example:

```
[hdfs@sandbox run]$ hadoop version
Hadoop 2.2.0.2.0.6.0-76
Subversion git@github.com:hortonworks/hadoop.git -r
8656b1cfad13b03b29e98cad042626205e7a1c86
Compiled by jenkins on 2013-10-18T00:19Z
Compiled with protoc 2.5.0
From source with checksum d23eeld271c6ac5bd27de664146be2
This command was run using /usr/lib/hadoop/hadoop-common-2.2.0.2.0.6.0-76.jar
```

Running the Pi Job

Command:

```
hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples-*.jar pi 10 10
```

Getting the Running Job List

Command:

```
yarn application -list
```

Example:

```
[yarn application -list
```

```
13/11/04 12:08:40 INFO client.RMProxy: Connecting to Resource Manager at
sandbox/10.11.2.159:8050
Total number of applications (application-types: [] and states: [SUBMITTED,
ACCEPTED, RUNNING]):1
```

User	Queue	Application-Id	Application-Name	Application-Type
		State	Final-State	Progress
		Tracking-URL		
application_1383594295029_0005			QuasiMonteCarlo	MAPREDUCE
hdfs	default	ACCEPTED	UNDEFINED	0%
		N/A		

Getting the Queue List

Command:

```
hadoop queue -list
```

Example:

```
[hdfs@sandbox run]$ hadoop queue -list
DEPRECATED: Use of this script to execute mapred command is deprecated.
Instead use the mapred command for it.

13/10/31 14:07:55 INFO client.RMProxy: Connecting to Resource Manager at
sandbox.hortonworks.com/10.0.2.15:8050
=====
Queue Name : default
Queue State : running
Scheduling Info : Capacity: 100.0, MaximumCapacity: 100.0, CurrentCapacity: 0.
0
```

Getting the Access Control List for the Current User

Command:

```
hadoop queue -showacls
```

Example:

```
root@a2nn:~> hadoop queue -showacls
Queue acls for user : root

Queue Operations
=====
default submit-job,administer-jobs
data-analysis submit-job,administer-jobs
```

Kill an Application

Command:

```
yarn application -kill <application_id>
```

Kill a Task

Command:

```
hadoop job -kill-task <task-id>
```

Fail a Task

Command:

```
hadoop job -fail-task <task-id>
```

List Attempts

Command:

```
hadoop job -list-attempt-ids <job-id> <task-type> <task-state>
```

1.3.2.9. MRv2 Troubleshooting Actions Checklist

1. Understand the issue. Check to see if the issue exists with all MRv2 jobs. Determine when a particular job or script last worked successfully. Understand the actual and expected behavior and formulate a problem statement.
2. Verify that all components related to MRv2 are running. You can use the `ps` and `jps` commands to check to see if the processes for dependent components are running. Ensure that all ports are listening, are bound to a process, and accept connection (i.e., firewall issues).
3. Look at the job details in the Resource Manager UI.
 - a. Use the UI to navigate to the job attempt.
 - b. Look at the log file for the failed attempt.
4. Look at the Resource Manager and the Node Manager log files in the Resource Manager UI, or on the specific nodes.
5. Use the `yarn logs` command to collect all of the logs of the Containers.
6. Check the Job Configuration in the Resource Manager UI to make sure that all of the desired parameters were actually passed on to the job.
7. Run the MRv2 `pi` job provided with the HDP examples to see if that job succeeds:
 - If it succeeds, check to see if there is a problem with the client or the data.
 - If it fails there is probably some basic problem with the process or the configuration.
8. If the job is run through streaming or pipes, run a similar job to troubleshoot.
9. If the job is started by one of the other HDP components, look at the component-specific guide.
10. Look for the operating system information and verify that it is supported.
11. Search the Hortonworks Knowledge Base for a possible solution.
12. If the issue is still not resolved, log a case in the Hortonworks Support Portal:

- a. Provide all of the information gathered in the preceding steps, along with the information in the “Checklist of Items to Collect” list in the following section.
- b. Tar the configuration files and the log files and attach them to the case.
- c. Inform Hortonworks if it is a Production, Development, or POC environment.

Checklist of Items to Collect

1. Collect the most recent log files for all of the MRv2 daemons.
2. Get copies of the following configuration files:
3. Provide the number of Data Nodes in the cluster, as well as the total number of nodes.
4. Use the `yarn logs` command to collect the log files of the Containers for all of the tasks.
5. How was HDP installed – with Ambari, or manually with RPM?
6. Provide hardware specifications: CPU, memory, disk drives, number of network interfaces.

1.3.2.10. Common Server-Side Issues

Resource Manager or Node Manager: Fails to Start or Crashes

Symptoms may include:

- Process appears to start, but then disappears from the process list.
- Node Manager cannot bind to interface.
- Kernel Panic, Halt.

Potential Root Cause: Existing Process Bound to Port

Troubleshooting Steps:

- Examine bound ports to ensure no other process has already bound.

Resolution Steps:

- Resolve the port conflict before attempting to restart the Resource Manager/Node Manager.

Information to Collect:

- List of bound interfaces/ports and the process.
- Resource Manager log.

Potential Root Cause: Incorrect File Permissions

Troubleshooting Steps:

- Verify that all Hadoop file system permissions are set properly.
- Verify the Hadoop configurations.

Resolution Steps:

- Follow the procedures for handling failure due to file permissions (see Hortonworks KB Solutions/Articles).
- Fix any incorrect configuration.

Information to Collect:

- Dump of file system permissions, ownership, and flags – by looking in the configuration value in the `yarn-site.xml` file for the `yarn.nodemanager.local-dirs` property. In this case, it has a value of `"/hadoop/yarn/local"`. From the command line, run:

```
ls -lR /hadoop/yarn/local
```

- Resource Manager log.
- Node Manager log.

Potential Root Cause: Incorrect Name-to-IP Resolution**Troubleshooting Steps:**

- Verify that the name/IP resolution is correct for all nodes in the cluster.

Resolution Steps:

- Fix any incorrect configuration.

Information to Collect:

- Local hosts file for all hosts on the system (`/etc/hosts`).
- Resolver configuration (`/etc/resolv.conf`).
- Network configuration (`/etc/sysconfig/network-scripts/ifcfg-ethX` where X = number of interface card).

Potential Root Cause: Java Heap Space Too Low**Troubleshooting Steps:**

- Examine the heap space property in `yarn-env.sh`
- Examine the settings in Ambari cluster management.

Resolution Steps:

**Potential Root Cause:
Permissions Not Set Correctly
on Local File System**

- Adjust the heap space property until the Resource Manager resumes running.

Information to Collect:

- `yarn-env.sh` from cluster.
- Screen-shot of Ambari cluster management mapred settings screen.
- Resource Manager log.
- Node Manager log.

Troubleshooting Steps:

- Examine the permissions on the various directories on the local file system.
- Verify proper ownership (yarn/mapred for MapReduce directories and hdfs for HDFS directories).

Resolution Steps:

- Use the `chmod` command to change the permissions of the directories to 755.
- Use the `chown` command to assign the directories to the correct owner (hdfs or yarn/mapred).
- Relaunch the Hadoop daemons using the correct user.

Information to Collect:

- `core-site.xml`, `hdfs-site.xml`, `mapred-site.xml`, `yarn-site.xml`
- Permissions listing for the directories listed in the above configuration files.

**Potential Root Cause:
Insufficient Disk Space****Troubleshooting Steps:**

- Verify that there is sufficient space on all system, log, and HDFS partitions.
- Run the `df -k` command on the Name/DataNodes to verify that there is sufficient capacity on the disk volumes used for storing NameNode or HDFS data.

Resolution Steps:

- Free up disk space on all nodes in the cluster.

-OR-

- Add additional capacity.

Information to Collect:

- Core dumps.
- Linux command: last (history).
- Dump of file system information.
- Output of `df -k` command.

Potential Root Cause: Reserved Disk Space is Set Higher than Free Space

Troubleshooting Steps:

- In `hdfs-site.xml`, check that the value of the `dfs.datanode.du.reserved` property is less than the available free space on the drive.

Resolution Steps:

- Configure an appropriate value, or increase free space.

Information to Collect:

- HDFS configuration files.

Node Java Process Exited Abnormally

Potential Root Cause: Improper Shutdown

Troubleshooting Steps:

- Investigate the OS history and the Hadoop audit logs.
- Verify that no edit log or fsimage corruption occurred.

Resolution Steps:

- Investigate the cause, and take measures to prevent future occurrence.

Information to Collect:

- Hadoop audit logs.
- Linux command: last (history).
- Linux user command history.

Potential Root Cause: Incorrect Memory Configuration

Troubleshooting Steps:

- Verify values in configuration files.

- Check logs for stack traces – out of heap space, or similar.

Resolution Steps:

- Fix configuration values and restart job/Resource Manager/Node Manager.

Information to Collect:

- Resource Manager.
- Node Manager.
- MapReduce v2 configuration files.

Node Manager Denied Communication with Resource Manager

Potential Root Cause: Hostname in Exclude File or Doesn't Exist in Include File

Troubleshooting Steps:

- Verify the contents of the files referenced in the `yarn.resourcemanager.nodes.exclude-path` property or the `yarn.resourcemanager.nodes.include-path` property.
- Verify that the host for this Node Manager is not being decommissioned.

Resolution Steps:

- If hostname for the Node Manager is in the file, and it is not meant for decommissioning, remove it.

Information to Collect:

- Files that are pointed to by the `yarn.resourcemanager.nodes.exclude-path` or `yarn.resourcemanager.nodes.include-path` properties.

Potential Root Cause: Node was Decommissioned and/or Reinserted into the Cluster

Troubleshooting Steps:

- This is a problem with HDFS. Refer to the HDFS Troubleshooting Guide.

Potential Root Cause: Resource Manager is Refusing the Connection

Troubleshooting Steps:

- Follow the steps (described previously) to ensure that the NameNode has started and is accepting requests.

Resolution Steps:

- Ensure that the DataNode is not in the "exclude" list.

- Ensure that the DataNode host is in the "include" list.

Information to Collect:

- NameNode slaves file.
- NameNode `hosts.deny` file (or the file specified as the blacklist in HDFS configuration).
- NameNode `hosts.allow` file (or the file specified as the whitelist in HDFS configuration).
- HDFS Configuration.

**Potential Root Cause:
NameNode is Refusing the
Connection**

Troubleshooting Steps:

- Follow the steps (described previously) to ensure that the NameNode has started and is accepting requests.

Resolution Steps:

- Ensure that the DataNode is not in the "exclude" list.

Information to Collect:

- NameNode slaves file.
- NameNode `hosts.deny` file (or the file specified as the blacklist in HDFS configuration).
- NameNode `hosts.allow` file (or the file specified as the whitelist in HDFS configuration).
- HDFS Configuration.

Error: Could Only be Replicated to x Nodes, Instead of n

**Potential Root Cause: At Least
One DataNode is Nonfunctional**

Troubleshooting Steps:

- This is a problem with HDFS. Refer to the HDFS Troubleshooting Guide.

**Potential Root Cause: One
or More DataNodes Are Out
of Space on Their Currently
Available Disk Drives**

Troubleshooting Steps:

- This is a problem with HDFS. Refer to the HDFS Troubleshooting Guide.

1.3.2.11. Common Client-Side Issues

Job Fails to Start

**Symptom: Exception When
Job Submitted, Potential Root**

Troubleshooting Steps:

Cause: Mistake in the Job's User Code

- Examine Node Manager/Resource Manager logs and task-logs to find the exact exception.

Resolution Steps:

- Examine the stack-trace for the thrown exception.
- Examine the user code to see if you can spot the error.

Information to Collect:

- Resource Manager log.
- Node Manager log.
- The exception trace that the user has mentioned, and the task logs.
- If possible, get at least a snippet of Java code from the area where the exception was thrown.

Symptom: "No Class Def Found" or Similar Exception When Trying to Start Job, Potential Root Cause - 1: Job's .jar File – or Other .jar File – Not on Classpath**Troubleshooting Steps:**

- Verify that the exception is `ClassNotFoundException`, `NoSuchMethodError`, or a similar exception.

Resolution Steps:

- Find the .jar file that contains the missing class and add it to the classpath.

Information to Collect:

- The entire command used to submit the job.
- The stack-trace from the Node Manager logs.

Potential Root Cause - 2: Main Class or Method of the Job Code is not "Public Static"**Troubleshooting Steps:**

- Examine the code for the main MRv2 class.

Resolution Steps:

- Set access modifiers to "public static"
- Recompile and re-test.

Information to Collect:

- The exact exception thrown by Hadoop.
- The job source code.

Job Seems to Hang in Setup

Symptom: Job Seems to Hang and Node Manager Becomes "Blacklisted", Potential Root Cause: Too Many Allowed Slots Configured for the System Memory on the Node

Troubleshooting Steps:

- Verify the amount of system memory.
- Calculate the required memory for each configured Container.
- Take into account any other processes running on the node.

Resolution Steps:

- Add all of the above. If the total is greater than the total available on the node, you will need to reduce the amount configured in the Container properties.

Symptom: Job Seems to Hang Without "Blacklisting", Potential Root Cause: No Node Managers Currently Available

Troubleshooting Steps:

- Verify the number of available MRv2 tasks available by looking at:

<Resource Manager host>:8088/cluster/nodes

Resolution Steps:

- Wait until more Node Managers become available, then see if the job runs.

Information to Collect:

- None until the job actually fails to run, then troubleshoot based on the failure symptom.

1.3.3. YARN Components

In Hadoop version 1, MapReduce was responsible for both processing and cluster resource management.

In Apache Hadoop version 2, cluster resource management has been moved from MapReduce into YARN, thus enabling other application engines to utilize YARN and Hadoop, while also improving MapReduce performance.

Use the following resources to learn more about YARN Components:

- [YARN Overview](#)
- [Resource Manager](#)
- [Node Manager](#)
- [Application Master](#)
- [Containers](#)

- [Job History Service](#)
- [YARN Walk-through](#)

1.3.3.1. YARN Overview

The MapReduce Distributed Processing Model

The MapReduce distributed processing model consists of a highly parallel map phase where input data is split into discrete chunks for processing. It is followed by the second and final reduce phase where the output of the map phase is aggregated to produce the desired result. The simple – and fairly restricted – nature of this programming model lends itself to very efficient implementations on an extremely large scale across thousands of cheap, commodity nodes.

Apache Hadoop MapReduce is the most popular open-source implementation of the MapReduce model. In particular, when MapReduce is paired with a distributed file system such as Apache Hadoop HDFS, which can provide very high aggregate I/O bandwidth across a large cluster, the economics of the system are extremely compelling – a key factor in the popularity of Hadoop.

One of the key features in MapReduce is the lack of data motion, i.e., move the computation to the data, rather than moving data to the compute node via the network. Specifically, the MapReduce tasks can be scheduled on the same physical nodes on which data resides in HDFS, which exposes the underlying storage layout across the cluster. This significantly reduces the network I/O patterns and keeps most of the I/O on the local disk or within the same rack, which is a key advantage.

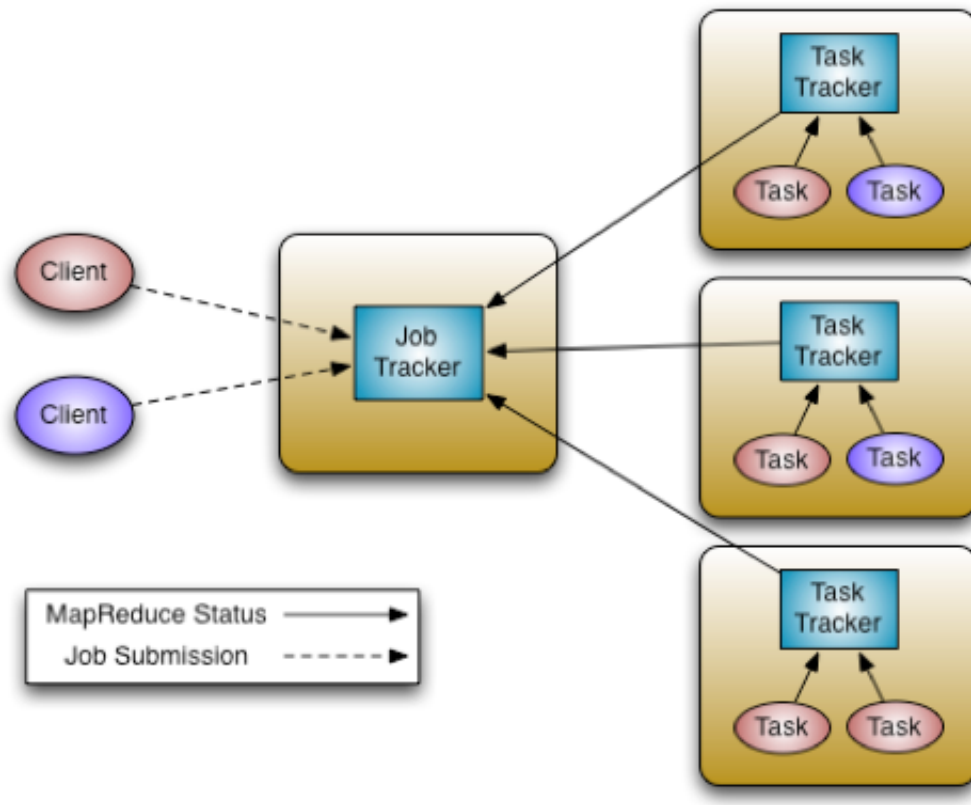
MapReduce on Hadoop Version 1

Apache Hadoop MapReduce is an open-source Apache Software Foundation project, and is an implementation of the MapReduce programming model described above. Version 1 of the Apache Hadoop MapReduce project can be broken down into the following major components:

- The end-user MapReduce API for programming the desired MapReduce application.
- The MapReduce framework, which is the run-time implementation of various phases such as the map phase, the sort/shuffle/merge aggregation, and the reduce phase.
- The MapReduce system, which is the back-end infrastructure required to run the MapReduce application, manage cluster resources, schedule thousands of concurrent jobs, etc.

This separation of elements offered significant benefits, particularly for end-users – they could completely focus on the application via the API, and allow the combination of the MapReduce Framework and the MapReduce System to deal with details such as resource management, fault-tolerance, scheduling, etc.

The version 1 Apache Hadoop MapReduce System was composed of the Job Tracker, which was the master, and the per-node slaves referred to as Task Trackers.



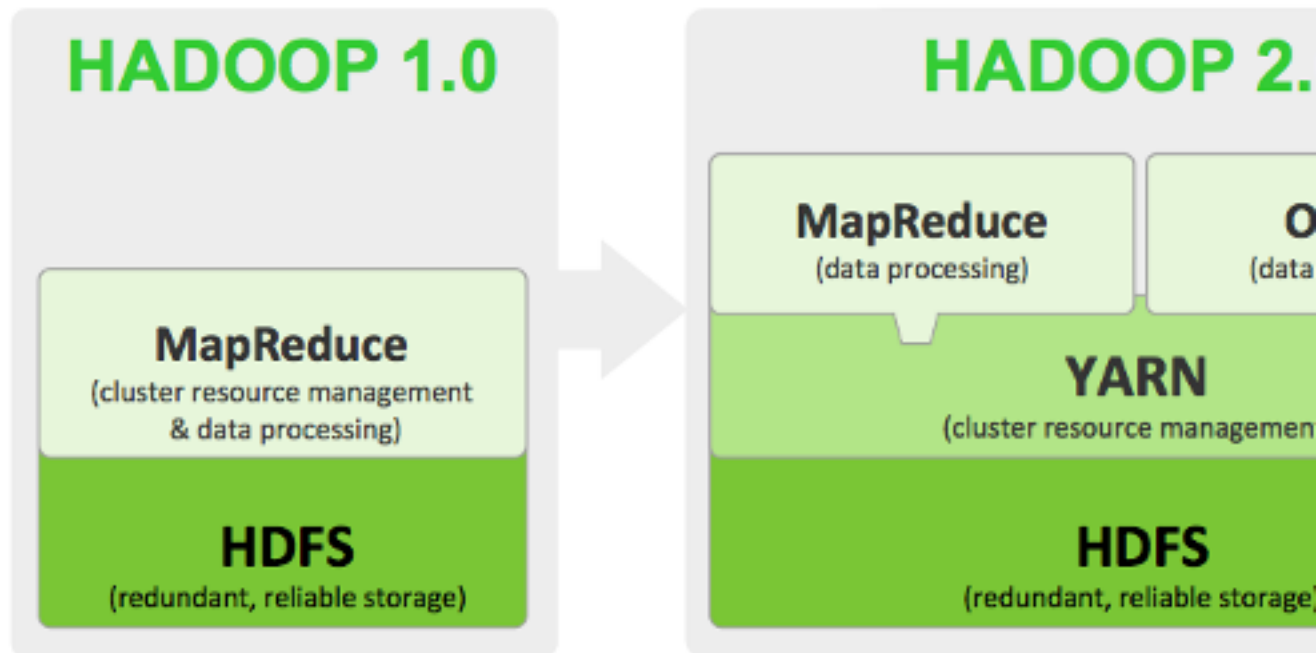
In Hadoop version 1, the Job Tracker is responsible for resource management (managing the worker nodes via the Task Trackers), tracking resource consumption and availability, and also job life-cycle management (scheduling individual tasks of a job, tracking progress, providing fault-tolerance for tasks, etc.).

The Task Tracker has simple responsibilities: executing launch and tear-down tasks on orders from the Job Tracker, and periodically providing task status information to the Job Tracker.

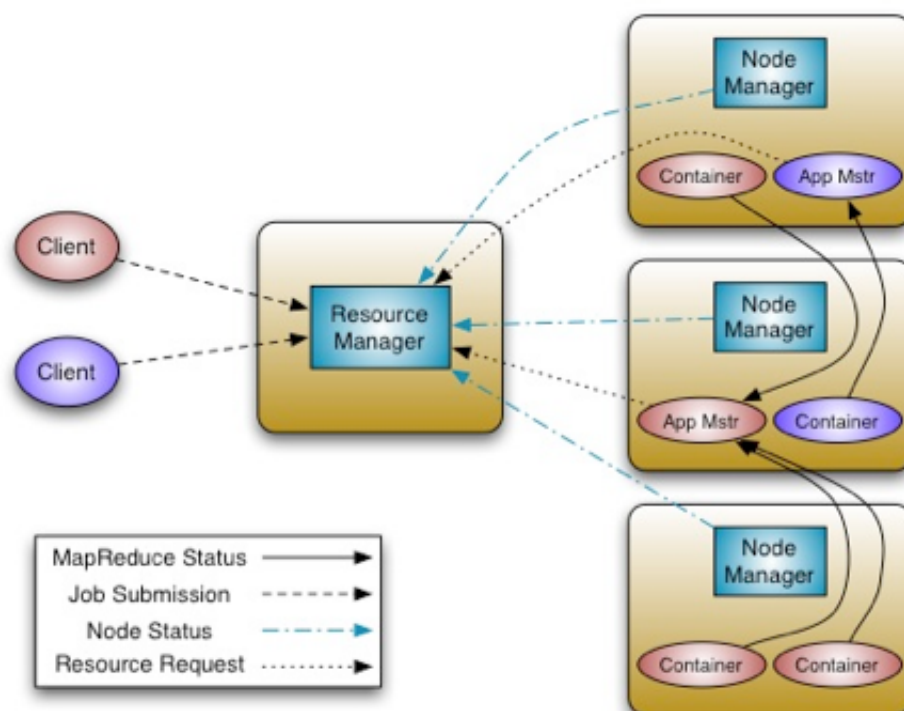
MapReduce works well for many applications, but not all. Other programming models are better suited for applications such as graph processing (Google Pregel, Apache Giraph) and iterative modeling using MPI (Message Parsing Interface). It also became clear that there was room for improvement in scalability, utilization, and user agility.

MapReduce on Hadoop YARN

In Apache Hadoop version 2, cluster resource management has been moved from MapReduce into YARN, thus enabling other application engines to utilize YARN and Hadoop. This also streamlines MapReduce to do what it does best: process data. With YARN, you can now run multiple applications in Hadoop, all sharing a common resource management.



The fundamental idea of YARN is to split up the two major responsibilities of the Job Tracker – resource management and job scheduling/monitoring – into separate daemons: a global Resource Manager and a per-application Application Master. The Resource Manager and its per-node slave, the Node Manager, form the new operating system for managing applications in a distributed manner.



The Resource Manager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application Application Master is a framework-specific entity, and is tasked with negotiating resources from the Resource Manager and working with the Node Managers to execute and monitor the component tasks.

The Resource Manager has a pluggable Scheduler component, which is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is a pure scheduler in the sense that it performs no monitoring or tracking of status for the application, offering no guarantees on restarting failed tasks either due to application or hardware failures. The scheduler performs its scheduling function based on the resource requirements of an application by using the abstract notion of a resource Container, which incorporates resource dimensions such as memory, CPU, disk, network, etc.

The Node Manager is the per-machine slave, and is responsible for launching the applications' Containers, monitoring their resource usage (CPU, memory, disk, network), and reporting this information to the Resource Manager.

The per-application Application Master has the responsibility of negotiating appropriate resource Containers from the Scheduler, tracking their status, and monitoring their progress. From the system perspective, the Application Master runs as a normal Container.

One of the crucial implementation details for MapReduce within the new YARN system is the reuse of the existing MapReduce framework without any major changes. This step was very important to ensure compatibility with existing MapReduce applications.

1.3.3.2. Resource Manager

The YARN Resource Manager is the master that arbitrates the available cluster resources, and thus helps manage the distributed applications running on the YARN system. It works together with the per-node Node Managers and the per-application Application Masters.

In YARN, the Resource Manager is primarily limited to scheduling, i.e., only arbitrating available resources in the system among the competing applications, and not concerning itself with per-application state management. The scheduler only handles an overall resource profile for each application, ignoring local optimizations and internal application flow. In fact, YARN completely departs from the static assignment of map and reduce slots because it treats the cluster as a resource pool. Because of this clear separation of responsibilities, the Resource Manager is able to address the important design requirements of scalability and support for alternate programming models.

The Resource Manager has a pluggable Scheduler component, which is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues, etc. The Scheduler is a pure scheduler in the sense that it performs no monitoring or tracking of status for the application, offering no guarantees on restarting failed tasks either due to application or hardware failures. The scheduler performs its scheduling function based on the resource requirements of an application by using the abstract notion of a resource Container, which incorporates resource dimensions such as memory, CPU, disk, network, etc. For more information about the Scheduler, see the [Capacity Scheduler](#) guide.

In contrast to many other workflow schedulers, the Resource Manager also has the ability to symmetrically request resources back from a running application. This situation typically

happens when cluster resources become scarce and the scheduler decides to revoke some (but not all) of the resources that were given to an application.

In YARN, Resource Requests can be strict or negotiable. This features provides Application Masters with a great deal of flexibility on how to fulfill the requests, e.g., by picking Containers to yield back that are less crucial for the computation, by check-pointing the state of a task, or by migrating the computation to other running Containers. Overall, this allows applications to preserve work, in contrast to platforms that kill Containers to satisfy resource constraints. If the application is non-collaborative, the Resource Manager can, after waiting a certain amount of time, obtain the needed resources by instructing the Node Managers to forcibly terminate Containers.

Resource Manager failures remain significant events affecting cluster availability. As of this writing, the Resource Manager will restart running Application Masters as it recovers its state. If the framework supports restart—and most will for routine fault tolerance—the platform will automatically restore users' pipelines.

In contrast to the Hadoop 1.0 Job Tracker, it is important to mention the tasks for which the Resource Manager is not responsible. While the Resource Manager will track application execution flow and task fault-tolerance, it will not provide access to the application status servlet, which is now part of the Application Master, nor will it track previously executed jobs, as that is now delegated to the Job History Service (a daemon running on a separate node). This is consistent with the view that the Resource Manager should only handle live resource scheduling, and helps YARN core components scale better than the Hadoop 1.0 Job Tracker.

1.3.3.3. Node Manager

The Node Manager manages the individual compute nodes in a Hadoop cluster. This includes keeping up-to-date with the Resource Manager, managing the life-cycle of application Containers, monitoring resource usage (memory, CPU) of individual Containers, monitoring node health, and managing logs and other auxiliary services that can be utilized by YARN applications.

On start-up, the Node Manager registers with the Resource Manager, and then sends heartbeats with its status and waits for instructions. Its primary goal is to manage application Containers assigned to it by the Resource Manager.

YARN Containers are described by a Container Launch Context (CLC). This record includes a map of environment variables, dependencies stored in remotely accessible storage, security tokens, payloads for Node Manager services, and the command necessary to create the process. After validating the authenticity of the Container lease, the Node Manager configures the environment for the Container, including initializing its monitoring subsystem with the resource constraints specified by the application. The Node Manager will also kill Containers as directed by the Resource Manager.

1.3.3.4. Application Master

The Application Master is the process that coordinates the execution of an application in the cluster. Each application has its own unique Application Master that is tasked with negotiating resources (Containers) from the Resource Manager and working with the Node Managers to execute and monitor the tasks. In the YARN design, MapReduce is just one

application framework; the design permits building and deploying distributed applications using other frameworks. For example, YARN ships with a Distributed Shell application that permits running a shell script on multiple nodes in a YARN cluster.

Once the Application Master is started (as a Container), it will periodically send heartbeats to the Resource Manager to affirm its health, and to update the record of its resource demands. After building a model of its requirements, the Application Master encodes its preferences and constraints in a heartbeat message to the Resource Manager. In response to subsequent heartbeats, the Application Master will receive a lease on Containers bound to an allocation of resources at a particular node in the cluster. Depending on the Containers it receives from the Resource Manager, the Application Master may update its execution plan to accommodate the excess or lack of resources. Container allocation/de-allocation can take place in a dynamic fashion as the application progresses.

1.3.3.5. Containers

A Container is a collection of physical resources on a single node, such as memory (RAM), CPU cores, and disks. There can be multiple Containers on a single Node (or a single large one). Every node in the system is considered to be composed of multiple Containers of minimum memory size (512MB or 1 GB, for example). The Application Master can request any Container as a multiple of the minimum memory size.

A Container thus represents a resource (memory, CPU) on a single node in a given cluster. A Container is supervised by the Node Manager and scheduled by the Resource Manager.

Each application starts out as an Application Master, which is itself a Container (often referred to as container -0). Once started, the Application Master must negotiate with the Resource Manager for more Containers. Container requests (and releases) can take place in a dynamic manner at run-time. For instance, a MapReduce job may request a certain amount of mapper Containers, and as they finish, release them and request that more reducer containers be started.

1.3.3.6. Job History Service

In Hadoop version 1, the Job Tracker was the master scheduler and tracker of all MapReduce jobs on the cluster. The new Resource Manager in Hadoop version 2, acting a scheduler, does not track job status. This task is now delegated to the JobHistoryService daemon, which is usually run on a separate node.

1.3.3.7. YARN Walk-through

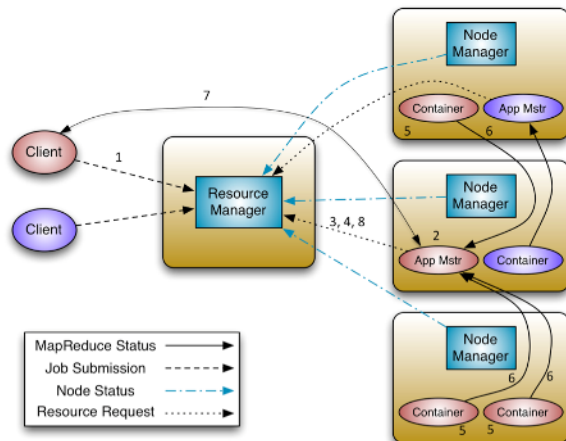
YARN application execution consists of the following steps:

- Application submission.
- Bootstrapping the Application Master instance for the application.
- Application execution managed by the Application Master instance.

Let's walk through an application execution sequence (steps are illustrated in the diagram):

1. A client program *submits* the application, including the necessary specifications to *launch the application-specific Application Master*.

2. The Resource Manager assumes responsibility for negotiating a specified Container in which to start the Application Master, and then *launches* the Application Master.
3. On boot-up, the Application Master *registers* with the Resource Manager. The registration allows the client program to query the Resource Manager for details, which allows it to directly communicate with its own Application Master.
4. During normal operation, the Application Master negotiates appropriate resource Containers via the resource-request protocol.
5. Upon successful Container allocations, the Application Master launches the Container by providing the Container launch specification to the Node Manager. The launch specification typically includes the necessary information to allow the Container to communicate with the Application Master.
6. The application code executing within the Container then provides necessary information (progress, status etc.) to its Application Master via an *application-specific protocol*.
7. During the application execution, the client that submitted the program communicates directly with the Application Master to get status, progress updates, etc. via an application-specific protocol.
8. Once the application is complete, the Application Master de-registers with the Resource Manager and shuts down, allowing its own Container to be repurposed.



1.3.4. Using Hadoop YARN

The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker i.e. resource management and job scheduling/monitoring, into separate daemons: a global **ResourceManager** and per-application **ApplicationMaster (AM)**.

The ResourceManager and per-node slave, the **NodeManager (NM)**, form the new, and generic, system for managing applications in a distributed manner. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system.

The per-application ApplicationMaster is, in effect, a framework specific entity and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the component tasks.

The ResourceManager has a pluggable **Scheduler**, which is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is a pure scheduler in the sense that it performs no monitoring or tracking of status for the application, offering no guarantees on restarting failed tasks either due to application failure or hardware failures. The Scheduler performs its scheduling function based on the resource requirements of the applications; it does so based on the abstract notion of a Resource Container which incorporates resource elements such as memory, CPU, disk, network etc. The NodeManager is the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager. The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress. From the system perspective, the ApplicationMaster itself runs as a normal container.

Use the following resources to learn more about YARN:

- [YARN architecture](#)
- [Writing YARN applications](#)
- [Capacity Scheduler](#)
- [Web Application Proxy](#)
- [Using YARN REST APIs](#)
- [Hadoop Auth](#)