Agile migration from MapReduce Version 1 to YARN for more robust YARN Hadoop clusters

Adam Kawa (kawa.adam@gmail.com)
Hadoop Developer, Administrator, and Instructor
Spotify and GetInData

The developerWorks tutorial "Agile migration of a single-node cluster from MapReduce Version 1 to YARN" guides you through the process of uninstalling MRv1 packages, installing YARN packages on a Hadoop cluster, starting necessary YARN daemons and configuring mandatory configuration settings to ensure that a cluster runs Java™ MapReduce jobs successfully. This tutorial expands the description of the YARN Hadoop cluster to include three more features that make Hadoop clusters more robust and functional.

The developerWorks tutorial, "Agile migration of a single-node cluster from MapReduce Version 1 to YARN," explains and demonstrates how small and frequent iterations help to smoothly migrate a single-node Hadoop cluster from MRv1 to YARN. This tutorial builds on that example and explains some features that make Hadoop clusters more robust and functional.

Migration environment

This migration example uses Ubuntu 12.04 with 4 GB of RAM, one 4-core CPU, and Hadoop YARN installed from Cloudera's distribution including Apache Hadoop (CDH).

Although the example uses CDH, you can use these steps on different distributions, including the standard Apache Hadoop or Hortonworks Data Platform 2.0 (HDP2). CDH was selected only because it provides Debian packages for MRv1 that were used to demonstrate the process of the migration from MRv1 to YARN on an Ubuntu server that is described in the previous tutorial.

The first tutorial uses the YARN preview from CDH4. This tutorial uses CDH5 because it contains a newer version of YARN.

Iterations 1, 2, and 3: Installing and smoke testing the YARN cluster

Iterations 1, 2, and 3 are described in the previous tutorial. After you complete these iterations, you have a single-node Hadoop cluster that runs all YARN daemons and processes sample MapReduce jobs. This cluster is extended in the next iterations, which are described in this tutorial.
Iteration 4. Configuring virtual cores

In the third iteration in Part 1, you configure multiple memory-related settings for YARN containers. In Iteration 4, you configure similar settings, but for virtual cores.

Step 1. Checking the number of virtual cores

YARN adds CPU as a resource by using a method similar to how it adds memory. You configure the NodeManagers to have a number of virtual cores (vcores). When an application requests a container, it asks for the number of vcores that a task needs.

When the NodeManager starts, it registers to the ResourceManager and specifies how much resource it has for running containers. Find this information by looking at the logs, as shown in Listing 1.

Listing 1. Checking a number of resources that NodeManager registers with to the ResourceManager

$ grep -r "Registered with ResourceManager" /var/log/hadoop-yarn/yarn-nodemanager-hakunamapdata.log | tail -n 1


The number of vcores available on the NodeManager is configured by using the yarn.nodemanager.resource.cpu-vcores parameter, which has a default value of 8. By default, a map or a reduce task requests one vcore in a container (specified with the mapreduce.map.cpu.vcores or mapreduce.reduce.cpu.vcores parameters according to the task); therefore, the NodeManager can run up to 8 MapReduce tasks.

Step 2. Adjusting the number of vcores

Typically, the number of virtual cores is set to be the same value as the number of physical cores on the node. Because this NodeManager runs on one 4-core CPU, set the number of virtual cores to 4 in the yarn-site.xml file, as shown in Listing 2.

Listing 2. Setting the number of virtual cores to 4

```
<property>
  <name>yarn.nodemanager.resource.cpu-vcores</name>
  <value>4</value>
</property>
```

Restart the NodeManager by issuing the command $ sudo /etc/init.d/hadoop-yarn-nodemanager restart.

Check to see whether the NodeManager re-registered with the updated number of vcores, as shown in Listing 3.
Listing 3. Check that NodeManager re-registered with the new number of vcores

```bash
$ grep -r "Registered with ResourceManager" /var/log/hadoop-yarn/yarn
-yarn-nodemanager-hakunamapdata.log | tail -n 1
```

INFO org.apache.hadoop.yarn.server.nodemanager.NodeStatusUpdaterImpl:
Registered with ResourceManager as hakunamapdata:44658 with total
resource of <memory:3072, vCores:4>

Step 3. Checking whether the vcore settings are recognized

To check whether the vcores are recognized, run a sample MapReduce job that starts several
tasks. Because the number of vcores is set to 4, up to three tasks can run concurrently (one vcore
is used by MapReduce ApplicationMaster). To run the sample job, issue the command $ hadoop
jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar pi 100 100.

Step 4. Requesting more vcores than the limit allows

It is a good idea to configure the limit for the maximum number of vcores that can be requested at
one time. This limit is configured by the `yarn.scheduler.maximum-allocation-vcores` parameter
and defaults to 32, a value that is larger than the typical number of vcores available on a
NodeManager.

Note: If an ApplicationMaster requests more vcores (or memory) than is available on any
NodeManager in a YARN cluster, the container is never granted by the ResourceManager and
the application halts. To demonstrate this behavior, run a tiny MapReduce job that requests many
vcores for its single map task, as shown in Listing 4.

Listing 4. Running a tiny MapReduce job

```bash
$ hadoop jar /usr/lib/hadoop-mapreduce/hadoop-mapreduce-examples.jar
pi -Dmapreduce.map.cpu.vcores=32 1 1
```

This code results in a MapReduce job that makes no progress for almost half an hour, as shown in
Figure 1.

Figure 1. MapReduce job that makes no progress for almost half an hour

According to the documentation, you can avoid this problem by specifying a value for
the `yarn.scheduler.maximum-allocation-vcores` that is not larger than the value for the
yarn.nodemanager.resource.cpu-vcores parameter on any NodeManager. Resource requests are capped at the maximum allocation limit and a container is eventually granted, as shown in Listing 5.

Listing 5. Configuration property that caps the resource requests to the maximum allocation limit

```xml
<property>
  <name>yarn.schedu.ter.maxim.ium-allocation-vcores</name>
  <value>3</value>
  <description>
    The maximum allocation for every container request at the RM, in terms of virtual CPU cores. Requests higher than this value don't take effect, and are capped to this value.
  </description>
</property>
```

Unfortunately, this feature currently **fails to work** as described and the requests are not capped at the limit. If a submitted application requests more vcores than the limits configured for every NodeManager, the containers are never granted, and no progress is made by the application (as shown in Figure 1.)

Iteration 5. Aggregating the application logs

In Iteration 5, you configure YARN to store logs (for example, stdout, stderr, and syslog) in HDFS instead of leaving them in predefined directories on local disks of individual worker nodes (as was the case with MRv1). This method gives you an easy way to access these logs for debugging purposes or for historical analyses to discover performance issues.

Step 1. Looking at logs from any task

Because log aggregation is disabled by default, you can't use a web UI to browse logs that are generated by tasks that run on a cluster after their application finishes.

If you click **logs** from the web UI, you should see a disappointing message that indicates that aggregation is not enabled, as shown in Figure 2.

**Figure 2. Error message: aggregation is not enabled**

![Error message: aggregation is not enabled. Try the nodemanager at hakunamapdata:43514](image)

This limitation makes the debugging of failed jobs inconvenient.

Step 2. Enabling log aggregation

Enable log aggregation by setting the `yarn.log-aggregation-enable` parameter to true in the yarn-site.xml file, as shown in Listing 6.
Listing 6. Enabling log aggregation

```xml
<property>
  <name>yarn.log-aggregation-enable</name>
  <value>true</value>
</property>
```

Step 3. Specifying location of log files in HDFS

Instead of using a default location (for example, `/tmp/logs`), specify, in the `yarn-site.xml` file, a dedicated location in HDFS where the aggregated logs are stored, as shown in Listing 7.

Listing 7. Specifying a dedicated location in HDFS where the aggregated logs are stored

```xml
<property>
  <name>yarn.nodemanager.remote-app-log-dir</name>
  <value>/app-logs</value>
  <description>Where to aggregate logs to.</description>
</property>
```

Step 4. Creating a directory in HDFS to store application logs

Create the directory (in Step 3) and set the appropriate permissions and ownership for it, as shown in Listing 8.

Listing 8. Creating a directory for aggregated logs and setting its permissions and ownership

```bash
$ sudo -u hdfs hadoop fs -mkdir /app-logs
$ sudo -u hdfs hadoop fs -chown -R yarn:hadoop /app-logs
$ sudo -u hdfs hadoop fs -chmod -R 777 /app-logs
```

(Optional) Step 5. Creating a custom suffix and compressing application logs

You might also want to configure the following parameters:

- `yarn.nodemanager.remote-app-log-dir-suffix`: Suffix that is appended to the remote log directory. Default is `logs`.
- `yarn.nodemanager.log-aggregation.compression-type`: Compression type that is used to compress aggregated logs. Default is `none`.

(Optional) Step 6. Enabling retention for application logs

When an application finishes, each NodeManager that runs at least one container for that application, aggregates all of the logs (that is, `stdout`, `stderr`, and `syslog`) from all of its local containers and writes them into a single, possibly compressed, file at a configured location in the HDFS, for example, `/app-logs`.

If you run many jobs and have many NodeManagers in the cluster, you definitely see many aggregated log files that are uploaded into HDFS. Enable an appropriate retention for the log files (as shown in Listing 9) to prevent the NameNode from running into trouble as a result of storing more files in HDFS, which causes increase in memory consumption for the HDFS metadata.
As shown in Listing 9, set the retention period to one month. On large clusters, a value such as 4 to 7 days might be more appropriate.

**Listing 9. Setting the retention period for application logs to one month**

```xml
<property>
  <name>yarn.log-aggregation.retain-seconds</name>
  <value>2629743</value>
</property>
```

You might also want to tweak the `yarn.log-aggregation.retain-check-interval-seconds` parameter to specify how often to check for application log files that qualify for retention.

**Step 7. Restarting necessary daemons**

To restart the NodeManager to enable the changes to take effect, issue the command:

```
$ sudo /etc/init.d/hadoop-yarn-nodemanager restart
```

If you used any non-default values for log retention, you must also restart the MapReduce Job History Server (MR JHS) because MR JHS is currently responsible for removing old application logs. Issue the following command to restart the server:

```
$ sudo /etc/init.d/hadoop-mapreduce-historyserver restart
```

**Step 8. Checking on application logs**

Submit any job to the cluster. After it finishes, check to see whether the logs appear in the web UI, as shown in Figure 3.

**Figure 3. MapReduce application logs available on web UI**
The logs are also available in HDFS in the directory /app-logs/<appOwner>/logs/<appId>, as shown in Listing 10.

**Listing 10. Listing application logs available in HDFS**

```
$ hadoop fs -ls /app-logs/kawaa/logs/application_1402783500700_0009
Found 1 items
-rw-r-----   3 kawaa hadoop     154331 2014-06-17 08:47 /app-logs/kawaa/logs/application_1402783500700_0009/hakunamapdata_59788
```

Because aggregated logs files are currently stored in TFile format, it is inconvenient to read them using traditional HDFS shell commands. Instead, use a dedicated CLI command that is supported by YARN, yarn logs, as shown in Listing 11.

**Listing 11. Using a CLI command to read application logs**

```
$ yarn logs
usage: general options are:
   -applicationId <arg>   ApplicationId
   -appOwner <arg>        AppOwner
   -containerId <arg>     ContainerId
   -nodeAddress <arg>     NodeAddress

$ yarn logs -applicationId application_1402783500700_0009 | less
```

**Step 9. Analyzing application logs by using a MapReduce application**

Use the open source tool Zlatanitor to parse application logs. Zlatanitor was mentioned at Hadoop Summit Europe 2014 during my talk about how to use Hadoop to analyze Hadoop.

**Iteration 6. Monitoring the health of the NodeManager**

In this iteration, configure the NodeManager to run a supplied script to determine whether a node is healthy. If a node is unhealthy, no further tasks are assigned to this node until it becomes healthy again. This approach makes it possible to temporarily exclude a problematic node from the cluster until a required action by an administrator is taken.

**Background**

NodeManagers (and the nodes they run on) can become unhealthy due to multiple software and hardware issues. Some examples include disk space issues, network problems, heavy memory swapping, high CPU contention, Hadoop misconfiguration, infrequent bugs in Hadoop or third-party libraries, and more.

Fortunately, Hadoop provides a simple mechanism by which administrators can configure the NodeManager to run a health-check script periodically to determine whether a node is healthy. If a node is reported as unhealthy, the ResourceManager blacklists a corresponding NodeManager and stops allocating containers on it. The NodeManager, however, still runs the script and if the node is reported as healthy again, the NodeManager is removed from the blacklist and starts processing tasks as usual.
Step 1. Implementing a health-check script that discovers basic network issues

Networking issues negatively affect the performance of HDFS and YARN because both systems transfer a significant amount of data between worker nodes.

Although the Hadoop speculative task execution runs can prevent a situation in which the whole pipeline of MapReduce jobs is slowed by a single slow or unhealthy node, it is expensive to run tasks redundantly. Furthermore, this feature works for MapReduce only. A YARN cluster, however, can now run many other types of workloads such as Tez, Spark, Impala, and Storm. These workloads might never support this feature. Therefore, it might be worth temporarily excluding the NodeManager that runs on a node with network problems until the issues are fixed.

For this example, use `ethtool` to diagnose possible network misconfigurations. Run `ethtool` with a single argument that specifies the Ethernet interface for which you want to see settings, as shown in Listing 12.

Listing 12. Running ethtool

```bash
$ ethtool eth0
Settings for eth0:
  ...                      
  Speed: 1000Mb/s
  Duplex: Full
  ...
```

Next, implement a simple bash script that examines the output of `ethtool` to check whether the network card is configured to run at the correct speed at full duplex, as shown in Listing 13.

Listing 13. Implementing a simple bash script that examines the output of ethtool

```bash
#!/bin/bash

NETWORK_INTERFACE=${1:-"eth0"}
INTERFACE_SPEED=${2:-"1000Mb/s"}
INTERFACE_DUPLEX=${3:-"Full"}

if ! $(ethtool $NETWORK_INTERFACE | grep -q "Speed: $INTERFACE_SPEED"); then
err_msg="ERROR: $NETWORK_INTERFACE does not run at $INTERFACE_SPEED speed."
fi

if ! $(ethtool $NETWORK_INTERFACE | grep -q "Duplex: $INTERFACE_DUPLEX"); then
err_msg="ERROR: $NETWORK_INTERFACE does not run at $INTERFACE_DUPLEX duplex. $err_msg"
fi

if [ ! -z "$err_msg" ]; then
  echo "$err_msg"
else
  echo "OK: NodeManager health check passes"
fi
```

Step 2. Testing a health-check script

Make sure that the script can be run by a yarn user, as shown in Listing 14.
Listing 14. Making sure that the script is executable by yarn user

```bash
$ chmod u+x /etc/hadoop/conf/nodemanager-health-checker.sh
$ chown yarn:hadoop /etc/hadoop/conf/nodemanager-health-checker.sh
```

Check to see if the script returns correct output, as shown in Listing 15.

Listing 15. Checking whether the health script returns the correct output

```bash
$ su yarn

$ /etc/hadoop/conf/nodemanager-health-checker.sh eth0 1000Mb/s Full
OK: NodeManager health check passes

$ /etc/hadoop/conf/nodemanager-health-checker.sh eth0 100Mb/s Ful
ERROR: eth0 does not run at full duplex. ERROR: eth0 does not run at 100Mb/s speed.
```

Step 3. Configuring the NodeManager to run the health checker script

Edit the yarn-site.xml file to configure the NodeManager to run this script periodically by providing a path to the script and a list of its parameters, as shown in Listing 16.

Listing 16. Edit yarn-site.xml to configure the NodeManager to run this health script periodically

```xml
<property>
    <name>yarn.nodemanager.health-checker.script.path</name>
    <value>/etc/hadoop/conf/nodemanager-health-checker.sh</value>
</property>

<property>
    <name>yarn.nodemanager.health-checker.script.opts</name>
    <value>eth0 1000Mb/s Full</value>
</property>
```

Restart the NodeManager to enable the changes to take effect, by using the command `sudo /etc/init.d/hadoop-yarn-nodemanager restart`.

Step 4. Smoke testing the health checker script

You can use `ethtool -s` to temporarily change the settings of your network device. To test the health checker script, you temporarily decrease the speed of your network card and set it to half duplex with the command `sudo ethtool -s eth0 speed 10 duplex half autoneg off`.

The health of the node is available in the ResourceManager web UI. If the node is unhealthy, the output of the script is displayed on the web UI. The health script runs every 10 minutes by default. This value is configured on the `yarn.nodemanager.health-checker.interval-ms` parameter.

Revert the changes by going back to the original settings by using the command `sudo ethtool -s eth0 speed 1000 duplex full autoneg on`.

You should see the NodeManager eventually becoming healthy again.
Step 5. Checking the health of local disks

According to the documentation, the health checker script is not supposed to return an error if only some of the local disks become bad. Instead, the NodeManager periodically checks the health of the local disks (specified on the nodemanager-local-dirs and nodemanager-log-dirs parameters) by itself. If the number of bad directories reaches a threshold (configured with yarn.nodemanager.disk-health-checker.min-healthy-disks), this information is sent to the ResourceManager, which marks a node as unhealthy, displays it on the web UI, and stops scheduling new containers on this node.

The default value of yarn.nodemanager.disk-health-checker.min-healthy-disks is 0.25. If less than 25 percent of local-dirs are healthy, the NodeManager is considered unhealthy.

To simulate the failure of a directory, change the ownership of a path from either nodemanager-local-dirs or nodemanager-log-dirs using the command $ sudo chown root:root /var/log/hadoop-yarn/userlogs

Assuming the default setting (yarn.nodemanager.disk-health-checker.interval-ms has the default value of 2 minutes) a bad directory is detected. The easiest way to notice a bad directory is to look at the ResourceManager web UI, as shown in Figure 4.

Figure 4. Bad directory that is automatically detected by the NodeManager and reported to the ResourceManager

You can also look at logs that are generated by the NodeManager that detects the bad directory, as shown in Listing 17.

Listing 17. Logs generated by the NodeManager that detects the bad directory

```
$ cat /var/log/hadoop-yarn/yarn-yarn-nodemanager-hakunamapdata.log
    | grep userlogs | tail -n 3
WARN ... Directory is not writable: /var/log/hadoop-yarn/userlogs, removing from the list of valid directories.
INFO ... Disk(s) failed. 1/1 log-dirs turned bad: /var/log/hadoop-yarn/userlogs
ERROR ... Most of the disks failed. 1/1 log-dirs turned bad: /var/log/hadoop-yarn/userlogs
```

You can look at logs from the ResourceManager that is notified by the NodeManager about its unhealthy status, as shown in Listing 18.
Listing 18. Logs from the ResourceManager that are notified by the NodeManager about its unhealthy status

$ cat /var/log/hadoop-yarn/yarn-yarn-resourcemanager-hakunamapdata.log | grep UNHEALTHY

... INFO ... Node hakunamapdata:40239 reported UNHEALTHY with details:
1/1 log-dirs turned bad: /var/log/hadoop-yarn/userlogs

Next, set the ownership of the bad directory back to yarn:yarn and restart the NodeManager, as shown in Listing 19.

Listing 19. Setting the ownership of a bad directory back to yarn:yarn

$ sudo chown yarn:yarn /var/log/hadoop-yarn/userlogs
$ sudo /etc/init.d/hadoop-yarn-nodemanager restart

Step 6. Extend the functionality of the health script

Obviously, the function of the health script can be further extended. For example, apart from diagnosing network misconfiguration, you can also use this script to blacklist the NodeManager that spends too much time performing Java garbage collection.

Iteration N: Adding more features

The YARN cluster is powerful, but you can extend its function by adding features. A complete description of all of them is outside the scope of this tutorial, but a few features are highlighted here:

1. Configure and tweak a scheduler of your choice, such as the capacity scheduler or the fair scheduler.
2. Enable or tweak multiple configuration settings that are related to MapReduce jobs, such as uberization or compression.
3. Enable the application recovery after the restart of the ResourceManager.
4. Take advantage of high availability for the ResourceManager.
5. Try other frameworks that are supported by YARN, such as Tez, Spark, Storm, and others.

Conclusion

This tutorial shows how to deploy multiple features to make a YARN cluster more reliable and functional. It explains how a feature works and how to adjust the configuration settings to properly configure it. The main focus, however, is to verify the validity of the newly deployed feature by looking at the web UI, the daemon logs, and the behavior of applications that are submitted to the cluster.
Resources

Learn

- Agile migration of a single-node cluster from MapReduce Version 1 to YARN (Adam Kawa, developerWorks, May 2014): Learn how to migrate a Hadoop cluster from MapReduce Version 1 (MRv1) to YARN in an agile way.
- Zlatanitor, a tool to parse application logs, was mentioned at Hadoop Summit Europe 2014 during the author’s talk about how to use Hadoop to analyze Hadoop.
- Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data: Learn details on two key IBM big data technologies.
- Visit the Apache Hadoop Project: The Apache website includes getting started information and download links.
- Using MapReduce and load balancing on the cloud (Kirpal A. Venkatesh, Kishorekumar Neelamegam, R. Revathy; developerWorks, July 2010): Learn how to implement the Hadoop MapReduce framework in a cloud environment and how to use virtual load balancing to improve the performance of both a single- and multiple-node system.

Get products and technologies

- Get Hadoop 0.20.1 from Apache.org.
- Use an open source tool, called Zlatanitor to parse application logs.
- Get Hadoop MapReduce.
- Get Hadoop HDFS.
- Innovate your next open source development project with IBM trial software, available for download or use in the cloud environment.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.
About the author

Adam Kawa

Adam Kawa works as a data engineer at Spotify, where his main responsibility is to maintain one of the largest Hadoop-YARN clusters in Europe. Every so often, he implements and troubleshoots MapReduce, Hive, Tez, and Pig applications. Adam also works as a Hadoop instructor at GetInData and is a frequent speaker at Hadoop conferences and Hadoop User Groups meetups. He co-organizes Stockholm and Warsaw Hadoop User Groups. Adam regularly blogs about the Hadoop ecosystem at HakunaMapData.com.

© Copyright IBM Corporation 2014

Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)