11 Running on a Spark standalone cluster

This chapter covers

- Components of Spark standalone cluster
- Spinning up the cluster
- Spark cluster Web UI
- Running applications
- Spark History Server
- Running on Amazon EC2

After describing common aspects of running Spark and examining Spark local modes in chapter 10, now we get to the first "real" Spark cluster type. The Spark standalone cluster is a Spark-specific cluster: it was built specifically for Spark, and it can't execute any other type of application. It's relatively simple and efficient and comes with Spark out of the box, so you can use it even if you don't have a YARN or Mesos installation.

In this chapter, we'll explain the runtime components of a standalone cluster and how to configure and control those components. A Spark standalone cluster

comes with its own web UI, and we'll show you how to use it to monitor cluster processes and running applications. A useful component for this is Spark's History Server; we'll also show you how to use it and explain why you should.

Spark provides scripts for quickly spinning up a standalone cluster on Amazon EC2. (If you aren't acquainted with it, Amazon EC2 is Amazon's cloud service, offering virtual servers for rent.) We'll walk you through how to do that. Let's get started.

11.1 Spark standalone cluster components

A standalone cluster comes bundled with Spark. It has a simple architecture and is easy to install and configure. Because it was built and optimized specifically for Spark, it has no extra functionalities with unnecessary generalizations, requirements, and configuration options, each with its own bugs. In short, the Spark standalone cluster is simple and fast.

The standalone cluster consists of master and worker (also called *slave*) processes. A master process acts as the cluster manager, as we mentioned in chapter 10. It accepts applications to be run and schedules worker resources (available CPU cores) among them. Worker processes launch application executors (and the driver for applications in cluster-deploy mode) for task execution. To refresh your memory, a driver orchestrates and monitors execution of Spark jobs, and executors execute a job's tasks.

Both masters and workers can be run on a single machine, essentially becoming Spark in local cluster mode (described in chapter 10), but this isn't how a Spark standalone cluster usually runs. You normally distribute workers across several nodes to avoid reaching the limits of a single machine's resources.

Naturally, Spark has to be installed on all nodes in the cluster in order for them to be usable as slaves. Installing Spark means unpacking a binary distribution or building your own version from Spark source files (for details, please see the official documentation at http://spark.apache.org/docs/latest/building-spark.html).

Figure 11.1 shows an example Spark standalone cluster running on two nodes with two workers:

Step 1. A client process submits an application to the master.
Step 2. The master instructs one of its workers to launch a driver.
Step 3. The worker spawns a driver JVM.
Step 4. The master instructs both workers to launch executors for the application.
Step 5. The workers spawn executor JVMs.
Step 6. The driver and executors communicate independent of the cluster's processes.

Each executor has a certain number of threads (CPU cores) allocated to it, which are task slots for running multiple tasks in parallel. In a Spark standalone cluster, for each application, there can be only one executor per worker process. If you need more executors per machine, you can start multiple worker processes. You may want to do this if your JVM heap is really large (greater than 64 GB) and GC is starting to affect job performance.

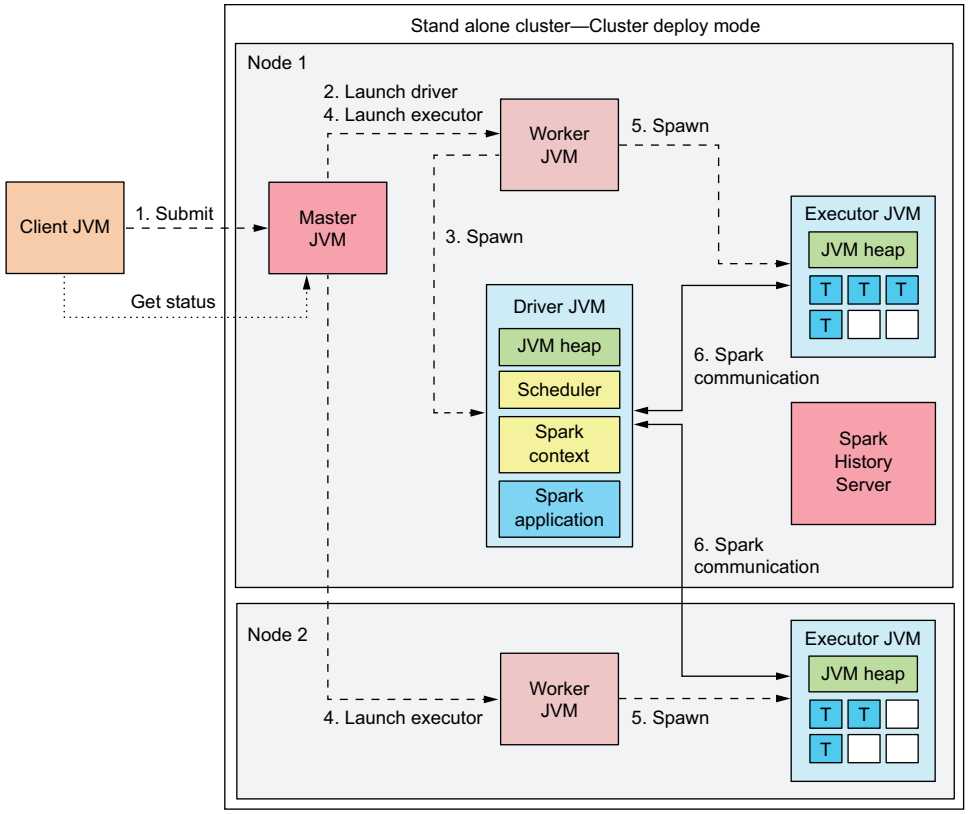

Figure 11.1 A Spark standalone cluster with an application in cluster-deploy mode. A master and one worker are running on Node 1, and the second worker is running on Node 2. Workers are spawning drivers' and executors' JVMs.

The driver in figure 11.1 is running in the cluster or, in other words, in cluster-deploy mode. As we said in chapter 10, it can also run in the client JVM, which is called *client-deploy mode*.

We should also mention that only one application is shown running in this cluster. If there were more, each would have its own set of executors and a separate driver running either in the cluster, like this one, or in its client's JVM (depending on the deploy mode).

An optional History Server is also shown in figure 11.1. It's used for viewing the Spark web UI after the application has exited. We'll explain this feature in more detail in section 11.5.

11.2 Starting the standalone cluster

Unlike starting Spark in one of the local cluster modes you saw in chapter 10, you must start a Spark standalone cluster before submitting an application or prior to starting the Spark shell. When the cluster is running, connect your application to the

cluster using the master connection URL. A master connection URL for a standalone cluster has the following syntax:

```
spark://master_hostname:port
```

If you have a standby master process running (see section 11.2.4), you can specify several addresses:

```
spark://master1_hostname:port1,master2_hostname:port2
```

To start the standalone cluster, you have two basic options: use Spark-provided scripts or start the components manually.

11.2.1 Starting the cluster with shell scripts

Startup scripts are the most convenient way to start Spark. They set up the proper environment and load your Spark default configuration. In order for scripts to run correctly, Spark should be installed at the same location on all the nodes in the cluster.

Spark provides three scripts for starting standalone-cluster components (you can find them in the *SPARK_HOME*/sbin directory):

- `start-master.sh` starts the master process.
- `start-slaves.sh` starts all defined worker processes.
- `start-all.sh` starts both master and worker processes.

Counterpart scripts for stopping the processes are also available: `stop-master.sh`, `stop-slaves.sh`, and `stop-all.sh`.

> **NOTE** On a Windows platform, no scripts for starting and stopping a standalone cluster are provided. The only option is to start and stop the cluster manually, as explained in section 11.2.2.

START-MASTER.SH

`start-master.sh` starts the master process. It takes no arguments and shows just one line when started:

```
$ sbin/start-master.sh
starting org.apache.spark.deploy.master.Master, logging to log_file
```

You can use the log file that the script outputs to find the command used to start the master and the master's runtime messages. The default log file is *SPARK_HOME*/logs/spark-*username*-org.apache.spark.deploy.master.Master-1-*hostname*.out.

To customize the `start-master.sh` script, you can use the system environment variables listed in table 11.1. The best way to apply them is to put them in the spark-env.sh file in the conf folder. (If it doesn't exist, you can use the spark-env.sh.template as a starting point.) The Java parameters in table 11.2 can be specified in the `SPARK_MASTER_OPTS` variable in this format:

```
-Dparam1_name=param1_value -Dparam2_name=param2_value
```

Table 11.1 System environment variables affecting the behavior of the `start-master.sh` script

System environment variable	Description
SPARK_MASTER_IP	Hostname the master should bind to.
SPARK_MASTER_PORT	Port the master should bind to (default is 7077).
SPARK_MASTER_WEBUI_PORT	Port on which the cluster web UI (described in section 11.4.3) should be started.
SPARK_DAEMON_MEMORY	Amount of heap memory to give the master and worker Java processes (default is 512 MB). The same parameter applies both to worker and master processes. Note that this only affects cluster daemon processes and not the driver or executors.
SPARK_MASTER_OPTS	Lets you pass additional Java parameters to the master process.

Table 11.2 Java parameters you can specify in the `SPARK_MASTER_OPTS` environment variable

Java parameter	Description
spark.deploy.defaultCores	Default maximum number of cores to allow per application. Applications can override this by setting the `spark.cores.max` parameter. If it isn't set, applications take all available cores on the machine.
spark.worker.timeout	Maximum number of seconds the master waits for a heartbeat from a worker before considering it lost (default is 60).
spark.dead.worker.persistence	Amount of time (measured as multiples of `spark.worker.timeout`) to keep dead workers displayed in the master web UI (default is 15).
spark.deploy.spreadOut	If set to `true`, which is the default, the master attempts to spread an application's executors across all workers, taking one core at a time. Otherwise, it starts an application's executors on the first free workers it finds, taking all available cores. Spreading out can be better for data locality when working with HDFS, because applications will run on a larger number of nodes, increasing the likelihood of running where data is stored.
spark.master.rest.enabled	Whether to start the standalone REST server for submitting applications (default is `true`). This is transparent to the end user.
spark.master.rest.port	Listening port for the standalone REST server (default is 6066).
spark.deploy.retainedApplications	Number of completed applications to display in the cluster web UI (default is 200).
spark.deploy.retainedDrivers	Number of completed drivers to display in the cluster web UI (default is 200).

In addition to the Java parameters in table 11.2, in the SPARK_MASTER_OPTS environment variable you can specify parameters for master recovery (which we'll describe in section 11.2.4).

START-SLAVES.SH

The start-slaves.sh script is a bit different. Using the SSH protocol, it connects to all machines defined in the *SPARK_HOME*/conf/slaves file and starts a worker process there. For this to work, Spark should be installed at the same location on all machines in the cluster.

A slaves file (similar to a Hadoop slaves file) should contain a list of worker hostnames, each on a separate line. If there are any duplicates in the file, starting additional workers on those machines will fail due to port conflicts. If you need more workers per machine, you can start them manually; or you can set the SPARK_WORKER_INSTANCES environment variable to the number of workers you want on each machine, and the start-slaves.sh script will start all of them automatically.

By default, the script will try to start all workers in tandem. For this, you need to set up *password-less* SSH. You can override this by defining any value for the SPARK_SSH_FOREGROUND environment variable. In that case, the script will start workers serially and allow you to enter a password for each remote machine.

Similar to the start-master.sh script, start-slaves.sh prints the path to the log file for each worker process it starts. The system environment variables listed in table 11.3 let you customize worker behavior.

Table 11.3 System environment variables affecting the behavior of worker processes

System environment variable	Description
SPARK_MASTER_IP	Hostname of the master process with which the worker should register.
SPARK_MASTER_PORT	Port of the master process with which the worker should register.
SPARK_WORKER_WEBUI_PORT	Port on which the worker web UI should be started (default is 8081).
SPARK_WORKER_CORES	Maximum combined number of CPU cores (task slots) for all executors launched by the worker.
SPARK_WORKER_MEMORY	Maximum combined total size of the Java heap for all executors launched by the worker.
SPARK_WORKER_DIR	Directory for the application's log files (and other application files, such as JAR files).
SPARK_WORKER_PORT	Port to which the worker should bind.
SPARK_WORKER_OPTS	Additional Java parameters to pass to the worker process.

The Java parameters specified in table 11.4 can be specified in the SPARK_WORKER_OPTS environment variable in this format:

```
-Dparam1_name=param1_value -Dparam2_name=param2_value
```

Table 11.4 Java parameters that can be specified in the SPARK_WORKER_OPTS environment variable

Java parameter	Description
spark.worker.timeout	Worker will be declared dead after this many seconds. Worker will send heartbeats to the master every spark.worker.timeout / 4 seconds.
spark.worker.cleanup.enabled	Interval for cleanup of old applications' log data and other files from the work directory (default is false).
spark.worker.cleanup.interval	Interval for cleanup of old applications' data (default is 30 minutes).
spark.worker.cleanup.appDataTtl	Time in seconds after which an application is considered old (default is 7 days, in seconds).

START-ALL.SH

The start-all.sh script calls start-master.sh and then start-slaves.sh.

11.2.2 Starting the cluster manually

Spark also provides an option to start cluster components manually, which is the only option available on a Windows platform. This can be accomplished by calling the spark-class script and specifying the complete Spark master or Spark worker class name as an argument. When starting the worker, you also need to specify the master URL. For example:

```
$ spark-class org.apache.spark.deploy.master.Master
$ spark-class org.apache.spark.deploy.worker.Worker spark://<IPADDR>:<PORT>
```

Both commands accept several optional parameters, listed in table 11.5. (Some of the parameters apply only to worker processes, as specified by the Only Worker column.) Starting the processes this way makes it a bit easier to specify these parameters, but starting with start-all.sh is still the most convenient method.

Table 11.5 Optional parameters that can be specified when starting masters or workers manually

Optional parameter	Description	Only worker
-h HOST or --host HOST	Hostname to listen on. For a master, the same as the SPARK_MASTER_HOST environment variable.	
-p PORT or --port PORT	Port to listen on (default is random). Same as SPARK_WORKER_PORT for a worker or SPARK_MASTER_PORT for a master.	
--webui-port PORT	Port for the web UI. Same as SPARK_MASTER_WEBUI_PORT or SPARK_WORKER_WEBUI_PORT. Default on a master is 8080 and on a worker is 8081.	

Table 11.5 Optional parameters that can be specified when starting masters or workers manually *(continued)*

Optional parameter	Description	Only worker
`--properties-file FILE`	Path to a custom Spark properties file (default is conf/spark-defaults.conf).	
`-c CORES` or `--cores CORES`	Number of cores to use. Same as the `SPARK_WORKER_CORES` environment variable.	P
`-m MEM` or `--memory MEM`	Amount of memory to use (for example, 1000 MB or 2 GB). Same as the `SPARK_WORKER_MEMORY` environment variable.	P
`-d DIR` or `--work-dir DIR`	Directory in which to run applications (default is `SPARK_HOME/work`). Same as the `SPARK_WORKER_DIR` environment variable.	P
`--help`	Shows help for invoking the script.	

11.2.3 Viewing Spark processes

If you're curious which cluster processes are started, you can use the JVM Process Status Tool (`jps` command) to view them. The `jps` command outputs PIDs and the names of JVM processes running on the machine:

```
$ jps
1696 CoarseGrainedExecutorBackend
403 Worker
1519 SparkSubmit
32655 Master
6080 DriverWrapper
```

Master and worker processes appear as `Master` and `Worker`. A driver running in the cluster appears as `DriverWrapper`, and a driver spawned by the `spark-submit` command (that also includes `spark-shell`) appears as `SparkSubmit`. Executor processes appear as `CoarseGrainedExecutorBackend`.

11.2.4 Standalone master high availability and recovery

A master process is the most important component in the standalone cluster. Because client processes connect to it to submit applications, the master requests resources from the workers on behalf of the clients, and users rely on it to view the state of running applications. If the master process dies, the cluster becomes unusable: clients can't submit new applications to the cluster, and users can't see the state of the currently running ones.

Master high availability means the master process will be automatically restarted if it goes down. Worker processes, on the other hand, aren't critical for cluster availability. This is the case because if one of the workers becomes unavailable, Spark will restart its tasks on another worker.

If the master process is restarted, Spark provides two ways to recover application and worker data that was running before the master died: using the filesystem and using ZooKeeper. ZooKeeper also provides automatic master high availability, as you'll see. With filesystem recovery, though, you have to set up master high availability yourself (if you need it) by using one of the tools available for that purpose (for example, start the master process from `inittab` with the `respawn` option[1]).

> **NOTE** All parameters mentioned in this section should be specified in the `SPARK_MASTER_OPTS` variable mentioned earlier and not in the spark-defaults.conf file.

FILESYSTEM MASTER RECOVERY

When using filesystem master recovery, a master persists information about registered workers and running applications in the directory specified by the `spark.deploy.recoveryDirectory` parameter. Normally, if the master restarts, workers re-register automatically, but the master loses information about running applications. This doesn't affect the applications, but users won't be able to monitor them through the master web UI.

If filesystem master recovery is enabled, the master will restore worker state instantly (no need for them to re-register), along with the state of any running applications. You enable filesystem recovery by setting the `spark.deploy.recoveryMode` parameter to `FILESYSTEM`.

ZOOKEEPER RECOVERY

ZooKeeper is a fast and simple system providing naming, distributed synchronization, and group services. ZooKeeper clients (or Spark, in this case) use it to coordinate their processes and to store small amounts of shared data. In addition to Spark, it's used in many other distributed systems.

ZooKeeper allows client processes to register with it and use its services to elect a leader process. Those processes not elected as leaders become followers. If a leader process goes down, a leader election process starts, producing a new leader.

To set up master high availability, you need to install and configure ZooKeeper. Then you start several master processes, instructing them to synchronize through ZooKeeper. Only one of them becomes a ZooKeeper leader. If an application tries to register with a master that currently isn't a leader, it will be turned down. If the leader fails, one of the other masters will take its place and restore the master's state using ZooKeeper's services.

Similar to filesystem recovery, the master will persist information about registered workers and running applications, but it will persist that information to ZooKeeper. This way, ZooKeeper provides both recovery and high-availability services for Spark master processes. To store the recovery data, ZooKeeper uses the directory specified by the `spark.deploy.zookeeper.dir` parameter on the machines where ZooKeeper is running.

---

[1] See the Linux man pages: www.manpages.info/linux/inittab.5.html.

You turn on ZooKeeper recovery by setting the `spark.deploy.recoveryMode` parameter to `ZOOKEEPER`. ZooKeeper needs to be accessible to the URLs specified by the `spark.deploy.zookeeper.url` parameter.

11.3 Standalone cluster web UI

When you start a master or a worker process, each starts its own web UI application. This is different than the way the web UI Spark context starts, as discussed in chapter 10. The Spark web UI shows information about applications, stages, tasks, and so on, and the standalone cluster web UI shows information about master and workers. An example master web UI is shown in figure 11.2.

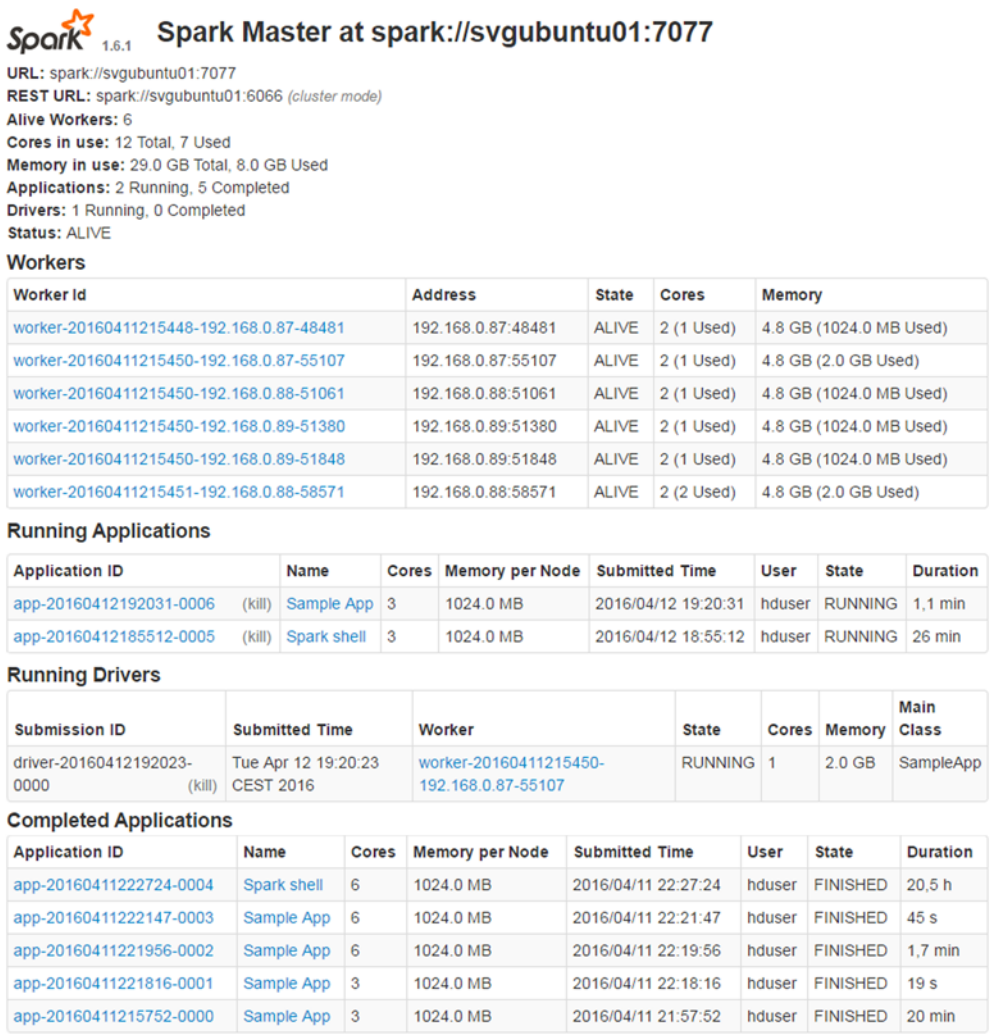

Figure 11.2 Example Spark master web UI page showing the running workers, running applications and drivers, and the completed applications and drivers

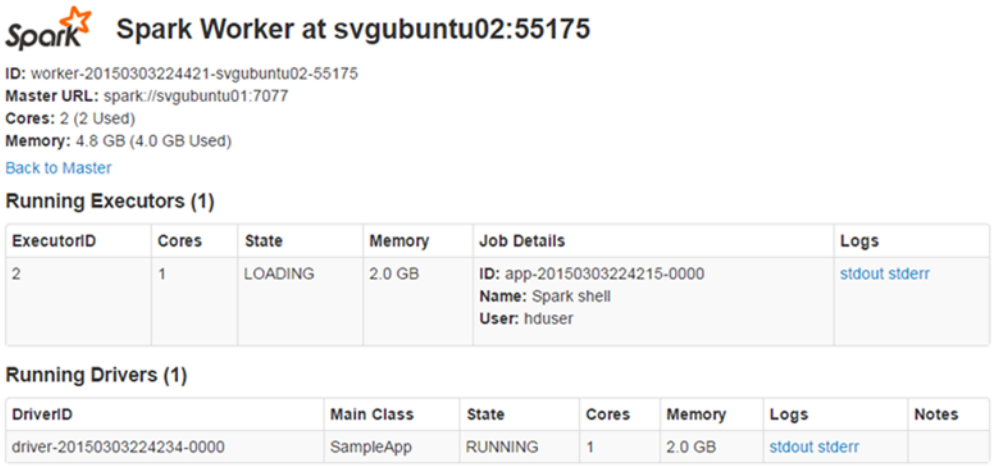

Figure 11.3 Sample Spark worker web UI

On the master web UI pages, you can see basic information about memory and CPU cores used and those available in the cluster, as well as information about workers, applications, and drivers. We'll talk more about these in the next section.

If you click a worker ID, you're taken to the web UI page started by a worker process. A sample UI page is shown in figure 11.3. On the worker web UI page, you can see which executors and drivers the worker is managing, and you can examine their log files by clicking the appropriate links.

If you click an application's name when on the master web UI page, you'll be taken to the Spark web UI page started by that application's Spark context. If you click an application's ID, though, you'll be taken to the application screen of the master web UI (figure 11.4).

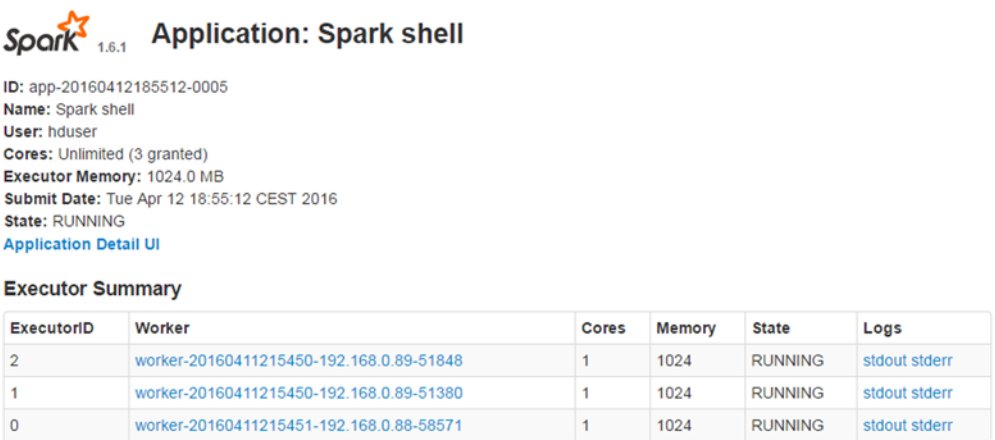

Figure 11.4 Sample Spark master web UI application screen

The application screen shows which workers and executors the application is running on. You can access the Spark web UI again by clicking the Application Detail UI link. You can also view the application's logs on each worker machine.

The Spark cluster web UIs (master and workers) and the Spark web UI come with Spark out of the box and offer a way of monitoring applications and jobs. That should be enough in most situations.

11.4 Running applications in a standalone cluster

As with the other cluster types, you can run Spark programs on a standalone cluster by submitting them with the `spark-submit` command, running them in a Spark shell, or instantiating and configuring a `SparkContext` object in your own application. We already talked about these options in chapter 10. In all three cases, you need to specify a master connection URL with the hostname and port of the master process.

> **NOTE** When connecting your applications to a Spark standalone cluster, it's important to use the exact hostname in the master connection URL as that used to start the master process (the one specified by the `SPARK_MASTER_IP` environment variable or your hostname).

You have two basic options when running Spark applications in a standalone cluster, and they differ in the location of the driver process.

11.4.1 Location of the driver

As we said in section 10.1.1, the driver process can run in the client process that was used to launch the application (like `spark-submit` script), or it can run in the cluster. Running in the client process is the default behavior and is equivalent to specifying the `--deploy-mode client` command-line argument. In this case, `spark-submit` will wait until your application finishes, and you'll see the output of your application on the screen.

> **NOTE** The `spark-shell` script supports only client-deploy mode.

To run the driver in the cluster, you have to specify the `--deploy-mode cluster` command-line argument. In that case, you'll see output similar to this:

```
Sending launch command to spark://<master_hostname>:7077
Driver successfully submitted as driver-20150303224234-0000
... waiting before polling master for driver state
... polling master for driver state
State of driver-20150303224234-0000 is RUNNING
Driver running on <client_hostname>:55175 (worker-20150303224421-
<client_hostname>-55175)
```

The option `--deploy-mode` is only used on standalone and Mesos clusters. YARN has a different master URL syntax.

> **NOTE** If you're embedding `SparkContext` in your application and you're not using the `spark-submit` script to connect to the standalone cluster, there's currently no way to specify deploy mode. It will default to client-deploy mode, and the driver will run in your application.

In cluster-deploy mode, the cluster manager takes care of the driver's resources and can automatically restart your application if the driver process fails (see section 11.4.5).

If you're submitting your application in cluster-deploy mode using the `spark-submit` script, the JAR file you specify needs to be available on the worker (at the location you specified) that will be executing the application. Because there's no way to say in advance which worker will execute your driver, you should put your application's JAR file on all the workers if you intend to use cluster-deploy mode, or you can put your application's JAR file on HDFS and use the HDFS URL as the JAR filename.

Log files from the driver running in cluster mode are available from the master and worker web UI pages. Of course, you can also access them directly on the filesystem of the corresponding worker.

> **NOTE** Python applications can't run in cluster-deploy mode on a standalone cluster.

The example web UI page in figure 11.2 (in section 11.3) shows three configured workers and two applications. One application is a Spark shell; the other is a custom application (called *Sample App*) submitted as a JAR file in cluster-deploy mode, so a running driver can also be seen on the web UI page. In cluster-deploy mode, the driver is spawned by one of the worker processes and uses one of its available CPU cores. We show this in figure 11.3 (in section 11.3).

11.4.2 Specifying the number of executors

Each of the two applications in figure 11.2 uses three cores out of six available (Sample App is using four cores; its driver is using the fourth one). This is accomplished by setting the parameter `spark.deploy.defaultCores` in the `SPARK_MASTER_OPTS` environment variable to 3 (as described previously). You can accomplish the same thing by setting the `spark.cores.max` parameter for each application you want to prevent from taking all available cores. You can also set the `SPARK_WORKER_CORES` environment variable to limit the number of cores each application can take *per machine*. If neither `spark.cores.max` nor `spark.deploy.defaultCores` were set, a single application would have taken all the available cores, and the subsequent applications would have had to wait for the first application to finish.

To control how many executors are allocated for your application, set `spark.cores.max` to the total number of cores you wish to use, and set `spark.executor.cores` to the number of cores per executor (or set their command-line equivalents: `--executor-cores` and `--total-executor-cores`). If you wish to use 3 executors with the total of 15 cores, set `spark.cores.max` to 15 and `spark.executor.cores` to 5. If you

plan to run only one application, leave these settings at the default value, which is *infinite* (`Int.MaxValue`).

11.4.3 Specifying extra classpath entries and files

In many situations, it's necessary to modify the classpath of your application or to make other files available to it. For example, your application may need a JDBC driver to access a relational database or other third-party classes not bundled with Spark. This means you need to modify the classpath of executor and driver processes, because that's where your application is executing. Special Spark parameters exist for these purposes, and you can apply them at different levels, as is often the case when configuring Spark.

> **NOTE** Techniques described in this section aren't specific to the standalone cluster and can be used on other cluster types, too.

USING THE SPARK_CLASSPATH VARIABLE

You can use the `SPARK_CLASSPATH` environment variable to add additional JAR files to the driver and executors. If you set it on the client machine, the extra classpath entries will be added to both the driver's and workers' classpaths. When using this variable, however, you'll need to manually copy the required files to the same location on all machines. Multiple JAR files are separated by semicolons (;) on Windows and by colons (:) on all other platforms.

USING THE COMMAND-LINE OPTIONS

Another option is to use the Spark configuration parameters `spark.driver.extraClassPath` and `spark.executor.extraClassPath` for JAR files, and `spark.driver.extraLibraryPath` and `spark.executor.extraLibraryPath` for native libraries. There are two additional `spark-submit` parameters for specifying driver paths: `--driver-class-path` and `--driver-library-path`. You should use these parameters if the driver is running in client mode, because then `--conf spark.driver.extraClassPath` won't work. JAR files specified with these options will be *prepended* to the appropriate executor classpaths. You'll still need to have these files on your worker machines as well.

USING THE –JARS PARAMETER

This option uses `spark-submit` with the `--jars` parameter, which automatically copies the specified JAR files (separated with commas) to the worker machines and adds them to the executor classpaths. That means the JAR files need not exist on the worker machines before submitting the application. Spark uses the same mechanism to distribute your application's JAR file to worker machines.

When using the `--jars` option, you can fetch the JAR files from different locations, depending on the prefix before the specified filename (a colon at the end of each prefix is required):

- `file:`—The default option described earlier. The file is copied to each worker.
- `local:`—The file exists on all worker machines at the exact same location.
- `hdfs:`—The file path is HDFS, and each worker can access it directly from HDFS.
- `http:`, `https:`, or `ftp:`—The file path is a URI.

NOTE If your application JAR includes classes or JARs also used by Spark itself, and you're experiencing conflicts among class versions, you can set the configuration parameter `spark.executor.userClassPathFirst` or `spark.driver.userClassPathFirst` to `true` to force Spark to load your classes before its own.

You can use a similar option (`--files`) to add ordinary files to workers (files that aren't JAR files or libraries). They can also be local, HDFS, HTTP, or FTP files. To use these files on workers, you need to access them with `SparkFiles.get(<filename>)`.

ADDING FILES PROGRAMMATICALLY

There is a programmatic method of adding JARs and files by calling `SparkContext`'s `addJar` and `addFile` methods. The `--jars` and `--files` options described earlier call these methods, so most things said previously also apply here. The only addition is that you can use `addFile(filename, true)` to recursively add an HDFS directory (the second argument means *recursive*).

ADDING ADDITIONAL PYTHON FILES

For Python applications, extra .egg, .zip, or .py files can be added with the `--py-files` `spark-submit` option. For example:

```
spark-submit --master <master_url> --py-files file1.py,file2.py main.py
```

where main.py is the Python file instantiating a Spark context.

11.4.4 Killing applications

If you submitted your application to the cluster in cluster mode, and the application is taking too long to complete or you want to stop it for some other reason, you can kill it by using the `spark-class` command like this:

```
spark-class org.apache.spark.deploy.Client kill <master_URL> <driver_ID>
```

You can do this only for applications whose driver is running in the cluster (cluster mode). For those applications submitted using the `spark-submit` command in client mode, you can kill the client process. You can still terminate particular stages (and jobs) using the Spark web UI, as described in section 10.4.2.

11.4.5 Application automatic restart

When submitting an application in cluster-deploy mode, a special command-line option (`--supervise`) tells Spark to restart the driver process if it fails (or ends abnormally). This restarts the entire application because it isn't possible to recover the state of a Spark program and continue at the point where it failed.

If the driver is failing every time and keeps getting restarted, you'll need to kill it using the method described in the previous section. Then you need to investigate the problem and change your application so that the driver executes without failing.

11.5 Spark History Server and event logging

We mentioned the History Server earlier. What's it for? Let's say you ran your application using `spark-submit`. Everything went smoothly, or so you thought. You suddenly notice something strange and would like to check a detail on the Spark web UI. You use the master's web UI to get to the application page, and you click the Application Detail UI link, but you get the message shown in figure 11.5. Or, even worse, you restarted the master process in the meantime, and your application isn't listed on the master's web UI.

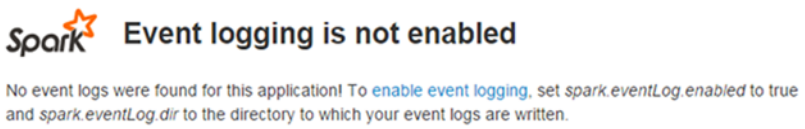

Figure 11.5 A web UI message showing that event logging isn't enabled

Event logging exists to help with these situations. When enabled, Spark logs events necessary for rendering the web UI in the folder specified by `spark.eventLog.dir`, which is /tmp/spark-events by default. The Spark master web UI will then be able to display this information in a manner identical to the Spark web UI so that data about jobs, stages, and tasks is available even after the application has finished. You enable event logging by setting `spark.eventLog.enabled` to `true`.

If you restarted (or stopped) the master, and your application is no longer available from the master web UI, you can start the Spark History Server, which displays a Spark web UI for applications whose events have been logged in the event log directory.

> **TIP** Unfortunately, if an application is killed before finishing, it may not appear in the History Server UI because the History Server expects to find a file named APPLICATION_COMPLETE in the application's directory (/tmp/spark-events/<application_id> by default). You can manually create an empty file with that name if it's missing, and the application will appear in the UI.

You start the Spark History Server with the script `start-history-server.sh` in the sbin directory, and you stop it with `stop-history-server.sh`. The default HTTP port is 18080. You can change this with the `spark.history.ui.port` parameter.

An example History Server page is shown in figure 11.6. Click any application ID link to go to the appropriate web UI pages, which we covered in section 11.4.

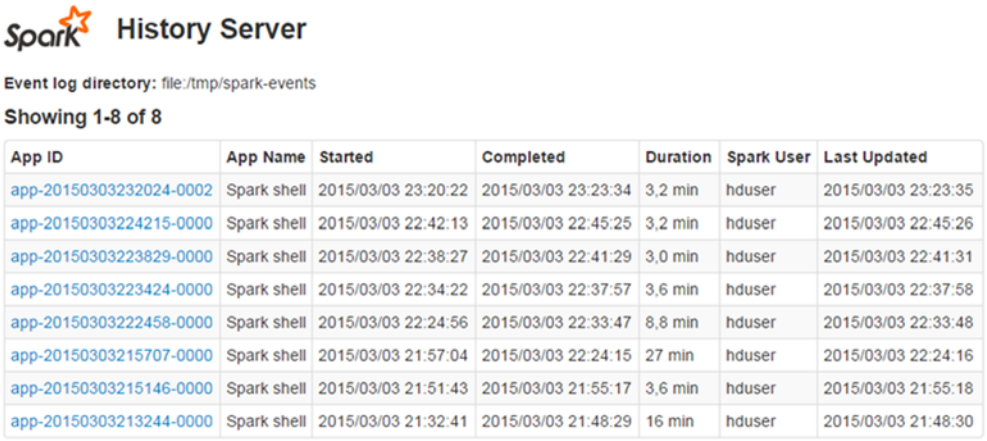

Figure 11.6 Spark History Server

You can customize the History Server with several environment variables: use `SPARK_DAEMON_MEMORY` to specify how much memory it should take, `SPARK_PUBLIC_DNS` to set its public address, `SPARK_DAEMON_JAVA_OPTS` to pass additional param-eters to its JVM, and `SPARK_HISTORY_OPTS` to pass to it `spark.history.*` parameters. For the complete list of these parameters, see the official documentation (http://spark.apache.org/docs/latest/configuration.html). You can also set `spark.history.*` parameters in the spark-default.conf file.

11.6 Running on Amazon EC2

You can use any physical or virtual machine to run Spark, but in this section, we'll show you how to use Spark's EC2 scripts to quickly set up a Spark standalone cluster in Amazon's AWS cloud. Amazon EC2 is Amazon's cloud service that lets you rent virtual servers to run your own applications. EC2 is just one of the services in Amazon Web Services (AWS); other services include storage and databases. AWS is popular because of its ease of use, broad set of features, and relatively low price. Of course, we don't want to start a flame war about which cloud provider is better. There are other providers and you can manually install Spark on them and set up a standalone cluster as described in this chapter.

To go through this tutorial, we'll use Amazon resources that go outside the free tier, so you should be prepared to spend a buck or two. You'll first obtain secret AWS keys, necessary for connecting to AWS services, and set up basic security. Then you'll use Spark's EC2 scripts to launch a cluster and log in to it. We'll show you how to stop and restart parts of the cluster you created. Finally, you'll destroy the fruit of all that hard work.

11.6.1 Prerequisites

In order to follow along, you should have an Amazon account and obtain these AWS keys: Access Key ID and Secret Access Key. These keys are used for user identification when using the AWS API. You can use the keys of your main user, but this isn't recommended. A better approach would be to create a new user with lower permissions and then generate and use those keys.

OBTAINING THE AWS SECRET KEYS

You create a new user with Amazon's Identity and Access Management (IAM) service. For the purposes of this tutorial, we created a user named *sparkuser* by selecting Services > IAM from the AWS landing page, going to the Users page, and clicking the Create New Users button. We entered a single username and left the option Generate an Access Key for Each User checked. Figure 11.7 shows our keys ready for download. You should store the keys in a safe place right away, because you won't be able to access them later.

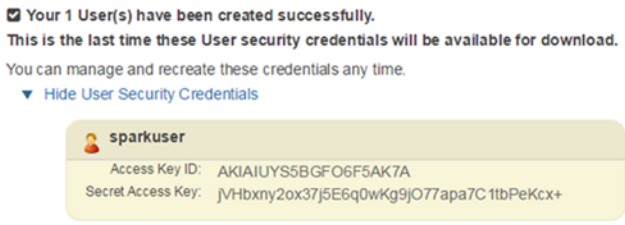

Figure 11.7 AWS user created

In order to successfully use this user for the Spark cluster setup, the user has to have adequate permissions. Click the new user's name, and then click the Attach Policy button on the Permissions tab (see figure 11.8). From the list of available policies, choose AmazonEC2FullAccess. This will be enough for your Spark setup.

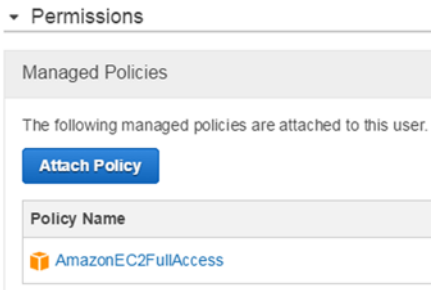

Figure 11.8 Giving the user adequate permissions

CREATING A KEY PAIR

The next prerequisite is a key pair, which is necessary for securing communication between a client and AWS services. On the EC2 Services page (available from any page through the top menu Services > EC2), under Network & Security, select Key Pairs and then choose your region in the upper-right corner. Choosing the correct region is important because keys generated for one region won't work in another. We chose Ireland (`eu-west-1`).

Click Create Key Pair, and give the pair a name. The name can be anything you like; we chose SparkKey. After creating the key pair, a private key will be automatically

downloaded as a <key_pair_name>.pem file. You should store that file in a secure but accessible place and change its access rights so that only you can read it:

```
chmod 400 SparkKey.pem
```

Just to be sure everything is OK before starting the scripts (you could have made a mistake in pasting its contents), check that the key is valid using this command:

```
openssl rsa -in SparkKey.pem -check
```

If the command outputs the contents of the file, all is well.

11.6.2 Creating an EC2 standalone cluster

Now let's look at the main `spark-ec2` script for managing the EC2 cluster. It used to come bundled with Spark, but has since moved to a separate project. To use it, create an ec2 directory in your SPARK_HOME folder and then clone the AMPLab's spark-ec2 GitHub repository (https://github.com/amplab/spark-ec2) into it. The spark-ec2 script has the following syntax:

```
spark-ec2 options action cluster_name
```

Table 11.6 shows the possible actions you can specify.

Table 11.6 Possible actions for the `spark-ec2` script

Action	Description
`launch`	Launches EC2 instances, installs the required software packages, and starts the Spark master and slaves
`login`	Logs in to the instance running the Spark master
`stop`	Stops all the cluster instances
`start`	Starts all the cluster instances, and reconfigures the cluster
`get-master`	Returns the address of the instance where the Spark master is running
`reboot-slaves`	Reboots instances where workers are running
`destroy`	An unrecoverable action that terminates EC2 instances and destroys the cluster

Depending on the `action` argument, you can use the same script to launch the cluster; log in to it; stop, start, and destroy the cluster; and restart the slave machines. Every action requires the `cluster_name` argument, which is used to reference the machines that will be created; and security credentials, which are the AWS secret keys and the key pair you created before. Options depend on the action chosen and will be explained as we go.

SPECIFYING THE CREDENTIALS

AWS secret keys are specified as the system environment variables `AWS_SECRET_ACCESS_KEY` and `AWS_ACCESS_KEY_ID`:

```
export AWS_SECRET_ACCESS_KEY=<your_AWS_access_key>
export AWS_ACCESS_KEY_ID=<your_AWS_access_key_id>
```

The key pair is specified with the `--key-pair` option (`-k` for short) containing the key pair name and the `--identity-file` option (`-i` for short) pointing to the `pem` file with the private key created earlier.

You also have to specify the `--region` option (`-r` for short) for all actions if you chose a region other than the default `us-east-1` as we did. Otherwise, the script won't be able to find your cluster's machines.

Up until this point, you have this (don't run this just yet)

```
spark-ec2 --key-pair=SparkKey --identity-file=SparkKey.pem \
--region=<your_region_if_dffrnt_than_us-east-1> launch spark-in-action
```

or the equivalent:

```
spark-ec2 -k SparkKey -i SparkKey.pem -r eu-west-1 launch spark-in-action
```

Running this command now would create a cluster called `spark-in-action`. But we'd like to change a few things before doing that.

CHANGING THE INSTANCE TYPES

Amazon offers many types of instances you can use for your VMs. These differ in number of CPUs, amount of memory available, and, of course, price.

The default instance type when creating EC2 instances with the `spark-ec2` script is `m1.large`, which has two cores and 7.5 GB of RAM. The same instance type will be used for the master and slave machines, which usually isn't desirable because the master is less hungry for resources. So we decided to use `m1.small` for the master. We also opted for `m1.medium` for slaves. The option for changing a slave's instance type is `--instance-type` (`-t` for short), and the option for changing the master is `--master-instance-type` (`-m` for short).

CHANGING THE HADOOP VERSION

You'll probably be using Hadoop on your EC2 instances. The default Hadoop version the `spark-ec2` script will install is 1.0.4, which may not be something you want. You can change that with the `--hadoop-major-version` parameter and set it to 2, which will install Spark prebuilt for Cloudera CDH 4.2.0, containing Hadoop 2.0.0 MR1.

CUSTOMIZING SECURITY GROUPS

By default, access to EC2 instances through internet ports isn't allowed. That prevents you from submitting applications to your Spark cluster running on EC2 instances directly from a client machine outside the EC2 cluster. EC2 security groups let you change inbound and outbound rules so that machines can communicate with the

Figure 11.9 Adding a custom security rule to allow access to the master instance from anywhere on the internet

outside world. The `spark-ec2` script sets up security groups to allow communication between the machines in your cluster, but access to port 7077 (the Spark standalone master default port) from the internet still isn't allowed. That's why you should create a security group (accessible from Services > EC2 > Security Groups > Create Security Group) with a rule as shown in figure 11.9 (we named it *Allow7077*).

Although opening a port to everybody like this isn't recommended, it's acceptable for short time periods in test environments. For production environments, we recommend that you restrict access to a single address.

You can assign a security group to all of your instances with the option `--additional-security-group`. Because you need the security group you just created only for the master (you don't need to access workers through port 7077), you won't use this option now and will add this security group manually to the master instance after your cluster is running.

LAUNCHING THE CLUSTER

The last thing you need to change is the number of slave machines. By default, only one will be created, and in this case you'd like more. The option for changing that is `--slaves` (`-s` for short).

This is the complete command:

```
./spark-ec2 --key-pair=SparkKey --identity-file=SparkKey.pem \
--slaves=3 --region=eu-west-1 --instance-type=m1.medium \
--master-instance-type=m1.small --hadoop-major-version=2 \
launch spark-in-action
```

After you start the script, it will create security groups, launch the appropriate instances, and install these packages: Scala, Spark, Hadoop, and Tachyon. The script downloads appropriate packages from online repositories and distributes them to workers using the `rsync` (remotely copy) program.

You can instruct the script to use an existing master instance and only create the slaves with the option `--use-existing-master`. To launch the cluster on an Amazon Virtual Private Cloud (VPC), you use the options `--vpc-id` and `--subnet-id`; everything else remains the same.

After the script is done, you'll see the new instances running in your EC2 console (figure 11.10). The cluster is now ready to be used.

11.6.3 Using the EC2 cluster

Now that you have your cluster, you can log in to see what your command created.

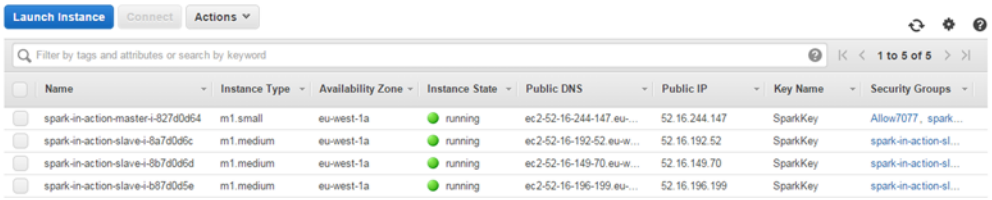

Figure 11.10 Spark cluster machines running on EC2

LOGGING IN

You can log in to your cluster in a couple of ways. First, the `spark-ec2` script provides an action for this:

```
spark-ec2 -k SparkKey -i SparkKey.pem -r eu-west-1 login spark-in-action
```

This will print out something similar to the following:

```
Searching for existing cluster spark-in-action...
Found 1 master(s), 3 slaves
Logging into master ec2-52-16-244-147.eu-west-1.compute.amazonaws.com

Last login: ...

       __|  __|_  )
       _|  (     /   Amazon Linux AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-ami/2013.03-release-notes/
There are 74 security update(s) out of 262 total update(s) available
Run "sudo yum update" to apply all updates.
Amazon Linux version 2014.09 is available.
root@ip-172-31-3-54 ~]$
```

And you're in! Another option is to add `ssh` directly to the public address of one of your instances, still using your private key (secret key environment variables aren't necessary in this case). You can find the addresses of your instances on the EC2 console. In this example, to log in to the master instance, you'd type

```
$ ssh -i SparkKey.pem root@52.16.171.131
```

If you don't feel like logging in to your EC2 console to find the address, you can use the `spark-ec2` script to obtain the master's hostname:

```
$ spark-ec2 -k SparkKey -i SparkKey.pem -r eu-west-1 login spark-in-action
Searching for existing cluster spark-in-action...
Found 1 master(s), 3 slaves
ec2-52-16-244-147.eu-west-1.compute.amazonaws.com
```

Then you can use that address for the `ssh` command.

CLUSTER CONFIGURATION

Upon logging in, you'll see that software packages were installed in the user's home directory. The default user is root (you can change that with the `--user` option), so the home directory is /root.

If you examine `spark-env.sh` in the spark/conf subdirectory, you'll see that the `spark-ec2` script added a few Spark configuration options. The ones we're most interested in are `SPARK_WORKER_INSTANCES` and `SPARK_WORKER_CORES`.

The number of instances per worker can be customized with the command-line option `--worker-instances`. Of course, you can manually alter the configuration in the `spark-env.sh` file, too. In that case, you should distribute the file to the workers.

`SPARK_WORKER_CORES` can't be customized using the `spark-ec2` script because the number of worker cores depends on the instance type selected. In the example, you used `m1.medium`, which has only one CPU, so the configured value was 1. For the default instance type (`m1.large` in the example), which has two CPUs, the configured default value is 2.

You'll also notice two Hadoop installations: `ephemeral-hdfs` and `persistent-hdfs`. Ephemeral HDFS is configured to use temporary storage available only while the machine is running. If the machine restarts, the ephemeral (temporary) data is lost.

Persistent HDFS is configured to use Elastic Block Store (EBS) storage, which means it won't be lost if the machine restarts. It also means keeping that data is going to cost you. You can add more EBS volumes to each instance with the `--ebs-vol-num`, `--ebs-vol-type`, and `--ebs-vol-size` options.

The spark-ec2 subdirectory contains the contents of the https://github.com/mesos/spark-ec2 GitHub repository. These are the actual scripts for setting up Spark EC2 clusters. One useful script there is `copy-dir`, enabling you to `rsync` a directory from one of the instances to the same path on all slave machines in a Spark configuration.

CONNECTING TO THE MASTER

If everything went smoothly, you should be able to start a Spark shell and connect to the master. For this, you should use the hostname returned by the `get-master` command and not its IP address.

If you assigned the Allow7077 security group to the master instance, you should be able to connect to the cluster from your client machine, too. Further configuration and application submission work as usual.

STOPPING, STARTING, AND REBOOTING

Stop and start actions obviously stop and start the entire cluster. After stopping, the machines will be in a stopped state, and data in temporary storage will be lost. After starting the cluster again, the necessary scripts will be called to rebuild the temporary data and repeat the cluster configuration. This will also cause any changes you may have made to your Spark configuration to be overwritten, although that data is kept in persistent storage. If you restart the machines using the EC2 console, your changes to your Spark configuration will be preserved.

spark-ec2 also lets you reboot slave machines. reboot-slaves is used similarly to other actions:

```
$ ./spark-ec2 -k SparkKey -i SparkKey.pem -r eu-west-1 \
reboot-slaves spark-in-action
Are you sure you want to reboot the cluster spark-in-action slaves?
Reboot cluster slaves spark-in-action (y/N): y
Searching for existing cluster spark-in-action...
Found 1 master(s), 3 slaves
Rebooting slaves...
Rebooting i-b87d0d5e
Rebooting i-8a7d0d6c
Rebooting i-8b7d0d6d
```

After a reboot, you'll have to run the start-slaves.sh script from the master machine, because slaves aren't started automatically.

11.6.4 Destroying the cluster

A destroy action (to use the AWS terminology) will terminate all cluster instances. Instances that are only stopped and not terminated may incur additional costs, although you may not be using them. For example, Spark instances use EBS permanent storage, and Amazon charges for EBS usage even though instances are stopped.

So it may be prudent to destroy the cluster if you won't be using it for a long time. It's straightforward to do:

```
spark-ec2 -k SparkKey -i SparkKey.pem destroy spark-in-action
Are you sure you want to destroy the cluster spark-in-action?
The following instances will be terminated:
Searching for existing cluster spark-in-action...
ALL DATA ON ALL NODES WILL BE LOST!!
Destroy cluster spark-in-action (y/N):
Terminating master...
Terminating slaves...
```

After this step, your only option is to launch another cluster (or pack up and go home).

11.7 Summary

- A standalone cluster comes bundled with Spark, has a simple architecture, and is easy to install and configure.
- It consists of master and worker processes.
- Spark applications on a Spark standalone cluster can run in cluster mode (the driver is running in the cluster) or client-deploy mode (the driver is running in the client JVM).
- You can start the standalone cluster with shell scripts or manually.
- The master process can be automatically restarted if it goes down, using filesystem master recovery or ZooKeeper recovery.

- The standalone cluster web UI gives useful information about running applications, master, and workers.
- You can specify extra classpath entries and files using the `SPARK_CLASSPATH` environment variable, using command-line options, using the `--jars` argument, and programmatically.
- The Spark History Server enables you to view the Spark web UI of applications that finished, but only if they were running while event logging was enabled.
- You can start a Spark standalone cluster on Amazon EC2 using the scripts in the Spark distribution.

12 Running on YARN and Mesos

This chapter covers

- YARN architecture
- YARN resource scheduling
- Configuring and running Spark on YARN
- Mesos architecture
- Mesos resource scheduling
- Configuring and running Spark on Mesos
- Running Spark from Docker

We examined a Spark standalone cluster in the previous chapter. Now it's time to tackle YARN and Mesos, two other cluster managers supported by Spark. They're both widely used (with YARN still more widespread) and offer similar functionalities, but each has its own specific strengths and weaknesses. Mesos is the only cluster manager supporting fine-grained resource scheduling mode; you can also use Mesos to run Spark tasks in Docker images. In fact, the Spark project was originally started to demonstrate the

usefulness of Mesos,[1] which illustrates Mesos's importance. YARN lets you access Kerberos-secured HDFS (Hadoop distributed filesystem restricted to users authenticated using the Kerberos authentication protocol) from your Spark applications.

In this chapter, we'll describe the architectures, installation and configuration options, and resource scheduling mechanisms for Mesos and YARN. We'll also highlight the differences between them and how to avoid common pitfalls. In short, this chapter will help you decide which platform better suits your needs. We'll start with YARN.

12.1 Running Spark on YARN

YARN (which, as you may recall, stands for yet another resource negotiator) is the new generation of Hadoop's MapReduce execution engine (for more information about MapReduce, see appendix B). Unlike the previous MapReduce engine, which could only run MapReduce jobs, YARN can run other types of programs (such as Spark). Most Hadoop installations already have YARN configured alongside HDFS, so YARN is the most natural execution engine for many potential and existing Spark users.

Spark was designed to be agnostic to the underlying cluster manager, and running Spark applications on YARN doesn't differ much from running them on other cluster managers, but there are a few differences you should be aware of. We'll go through those differences here.

We'll begin our exploration of running Spark on YARN by first looking at the YARN architecture. Then we'll describe how to submit Spark applications to YARN, then explain the differences between running Spark applications on YARN compared to a Spark standalone cluster.

12.1.1 YARN architecture

The basic YARN architecture is similar to Spark's standalone cluster architecture. Its main components are a *resource manager* (it could be likened to Spark's master process) for each cluster and a *node manager* (similar to Spark's worker processes) for each node in the cluster. Unlike running on Spark's standalone cluster, applications on YARN run in *containers* (JVM processes to which CPU and memory resources are granted). An *application master* for each application is a special component. Running in its own container, it's responsible for requesting application resources from the resource manager. When Spark is running on YARN, the Spark driver process acts as the YARN application master. Node managers track resources used by containers and report to the resource manager.

Figure 12.1 shows a YARN cluster with two nodes and a Spark application running in the cluster. You'll notice that this figure is similar to figure 11.1, but the process of starting an application is somewhat different.

A *client* first submits an application to the resource manager (step 1), which directs one of the node managers to allocate a container for the application master (step 2). The node manager launches a container (step 3) for the application master (Spark's

[1] See "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," by Benjamin Hindman et al., http://mesos.berkeley.edu/mesos_tech_report.pdf.

driver), which then asks the resource manager for more containers to be used as Spark executors (step 4). When the resources are granted, the application master asks the node managers to launch executors in the new containers (step 5), and the node managers obey (step 6). From that point on, driver and executors communicate independently of YARN components, in the same way as when they're running in other types of clusters. Clients can query the application's status at any time.

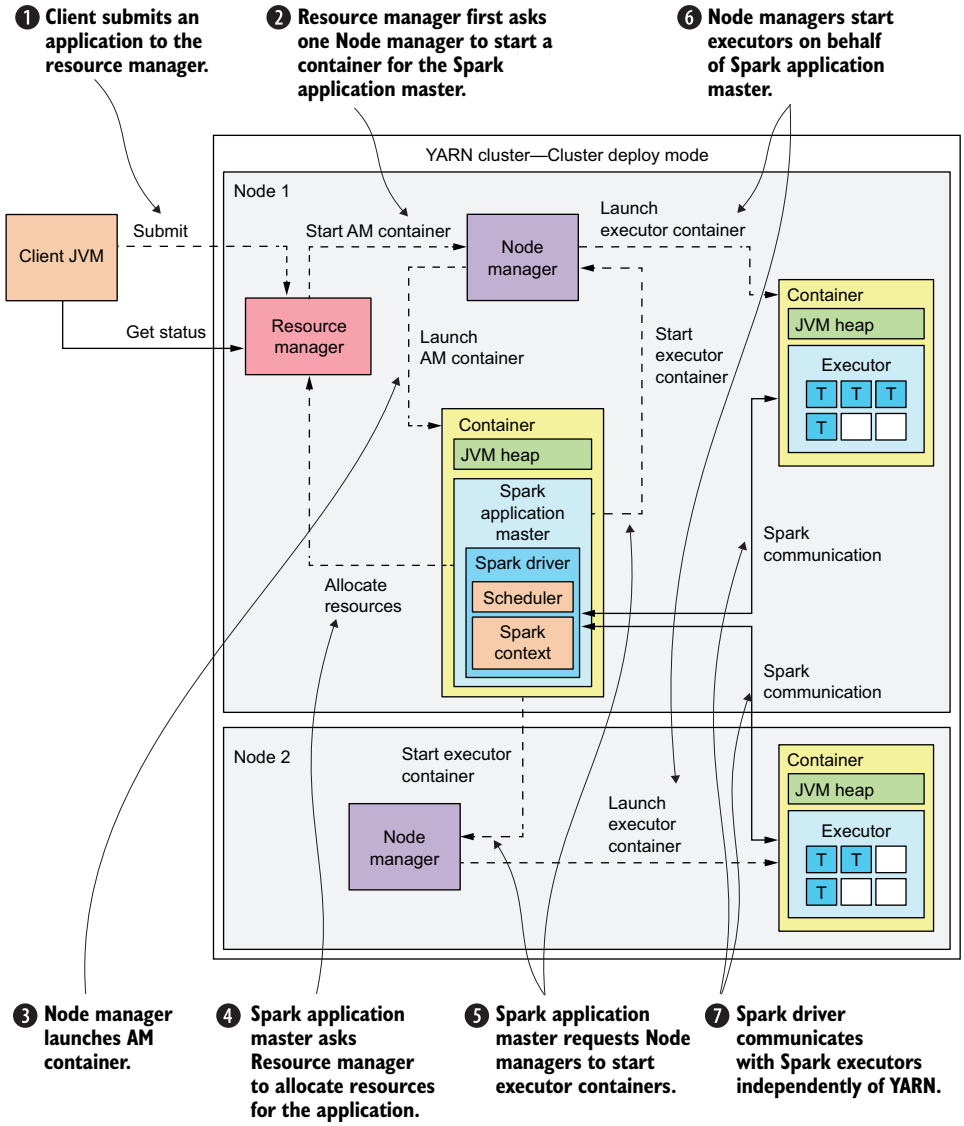

Figure 12.1 YARN architecture in an example cluster of two nodes. The client submits an application, whereby the resource manager starts the container for the application master (Spark driver). The application master requests more containers for Spark executors. Once the containers start, the Spark driver communicates directly with its executors.

Figure 12.1 shows only one application running in the cluster. But multiple applications can run in a single YARN cluster, be they Spark applications or applications of another type. In that case, each application has its own application master. The number of containers depends on the application type. At minimum, an application could consist of only an application master. Because Spark needs a driver *and* executors, it will always have a container for the application master (Spark driver) and one or more containers for its executors. Unlike Spark's workers, YARN's node managers can launch more than one container (executor) per application.

12.1.2 Installing, configuring, and starting YARN

This section contains an overview of YARN and Hadoop installation and configuration. For more information, we recommend *Hadoop in Practice, Second Edition*, by Alex Holmes (Manning, 2015) and *Hadoop: The Definitive Guide, Fourth Edition*, by Tom White (O'Reilly, 2015).

YARN is installed together with Hadoop. The installation is straightforward: from the Hadoop download page (https://hadoop.apache.org/releases.html), you need to download and extract Hadoop's distribution archive on every machine that is to be part of your cluster. Similar to Spark, you can use YARN in three possible modes:

- *Standalone (local) mode*—Runs as a single Java process. This is comparable to Spark's local mode, described in chapter 10.
- *Pseudo-distributed mode*—Runs all Hadoop daemons (several Java processes) on a single machine. This is comparable to Spark's local cluster mode, described in chapter 10.
- *Fully distributed mode*—Runs on multiple machines.

CONFIGURATION FILES

Hadoop's XML-based configuration files are located in the etc/hadoop directory of the main installation location. The main configuration files are as follows:

- *slaves*—List of hostnames (one per line) of the machines in the cluster. Hadoop's slaves file is the same as the slaves file in Spark's standalone cluster configuration.
- *hdfs-site.xml*—Configuration pertaining to Hadoop's filesystem.
- *yarn-site.xml*—YARN configuration.
- *yarn-env.sh*—YARN environment variables.
- *core-site.xml*—Various security, high-availability, and filesystem parameters.

Copy the configuration files to all the machines in the cluster. We'll mention specific configuration options pertinent to Spark in the coming sections.

STARTING AND STOPPING YARN

Table 12.1 lists the scripts for starting and stopping YARN and HDFS daemons. They're available in the sbin directory of the main Hadoop installation location.

Table 12.1 **Scripts for starting and stopping YARN and HDFS daemons**

Script file	What it does
`start-hdfs.sh` / `stop-hdfs.sh`	Starts/stops HDFS daemons on all machines listed in the slaves file
`start-yarn.sh` / `stop-yarn.sh`	Starts/stops YARN daemons on all machines listed in the slaves file
`start-all.sh` / `stop-all.sh`	Starts/stops both HDFS and YARN daemons on all machines listed in the slaves file

12.1.3 Resource scheduling in YARN

YARN's `ResourceManager`, mentioned previously, has a pluggable interface to allow different plug-ins to implement its resource-scheduling functions. There are three main scheduler plug-ins: the *FIFO scheduler*, the *capacity scheduler*, and the *fair scheduler*. You specify the desired scheduler by setting the property `yarn.resourcemanager.scheduler.class` in the yarn-site.xml file to the scheduler's class name. The default is the capacity scheduler (the value `org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.CapacityScheduler`).

These schedulers treat Spark like any other application running in YARN. They allocate CPU and memory to Spark according to their logic. Once they do, Spark schedules resources for its own jobs internally, as discussed in chapter 10.

FIFO SCHEDULER

The FIFO scheduler is the simplest of the scheduler plug-ins. It lets applications take all the resources they need. If two applications require the same resources, the first application that requests them will be first served (FIFO).

CAPACITY SCHEDULER

The capacity scheduler (the default scheduler in YARN) was designed to allow for sharing of a single YARN cluster by different organizations, and it guarantees that each organization will always have a certain amount of resources available (*guaranteed capacity*). The main unit of resources scheduled by YARN is a *queue*. Each queue's capacity determines the percentage of cluster resources that can be used by applications submitted to it. A hierarchy of queues can be set up to reflect a hierarchy of capacity requirements by organizations, so that sub-queues (sub-organizations) can share the resources of a single queue and thus not affect others. In a single queue, the resources are scheduled in FIFO fashion.

If enabled, capacity scheduling can be elastic, meaning it allows organizations to use any excess capacity not used by others. Preemption isn't supported, which means the excess capacity temporarily allocated to some organizations isn't automatically freed when demanded by organizations originally entitled to use it. If that happens, the "rightful owners" have to wait until the "guests" have finished using their resources.

FAIR SCHEDULER

The fair scheduler tries to assign resources in such a way that all applications get (on average) an equal share. By default, it bases its decisions on memory only, but you can configure it to schedule with both memory and CPU.

Like the capacity scheduler, it also organizes applications into queues. The fair scheduler also supports application priorities (some applications should get more resources than others) and minimum capacity requirements. It offers more flexibility than the capacity scheduler. It enables preemption, meaning when an application demands resources, the fair scheduler can take some resources from other running applications. It can schedule resources according to FIFO scheduling, fair scheduling, and dominant resource fairness scheduling. Dominant resource fairness scheduling takes into account CPU and memory (whereas in normal operation, only memory affects scheduling decisions).

12.1.4 Submitting Spark applications to YARN

As with running applications in a Spark standalone cluster, depending on where the driver process is running, Spark has two modes of running applications on YARN: the driver can run in YARN or it can run on the client machine. If you want it to run in a cluster, the Spark master connection URL should be as follows:

```
--master yarn-cluster
```

If you want it to run on the client machine, use:

```
--master yarn-client
```

> **NOTE** The spark-shell can't be started in yarn-cluster mode because an interactive connection with the driver is required.

Figure 12.1 shows an example of running Spark on YARN in cluster-deploy mode. Figure 12.2 shows Spark running on YARN in client-deploy mode. As you can see, the order of calls is similar to cluster-deploy mode. What is different is that resource allocation and internal Spark communications are now split between Spark's application master and the driver. Spark's application master handles resource allocation and communication with the resource manager, and the driver communicates directly with Spark's executors. Communication between the driver and the application master is now necessary, too.

Submitting an application to YARN in client mode is similar to the way it's done for a standalone cluster. You'll see the output of your application in the client window. You can kill the application by stopping the client process.

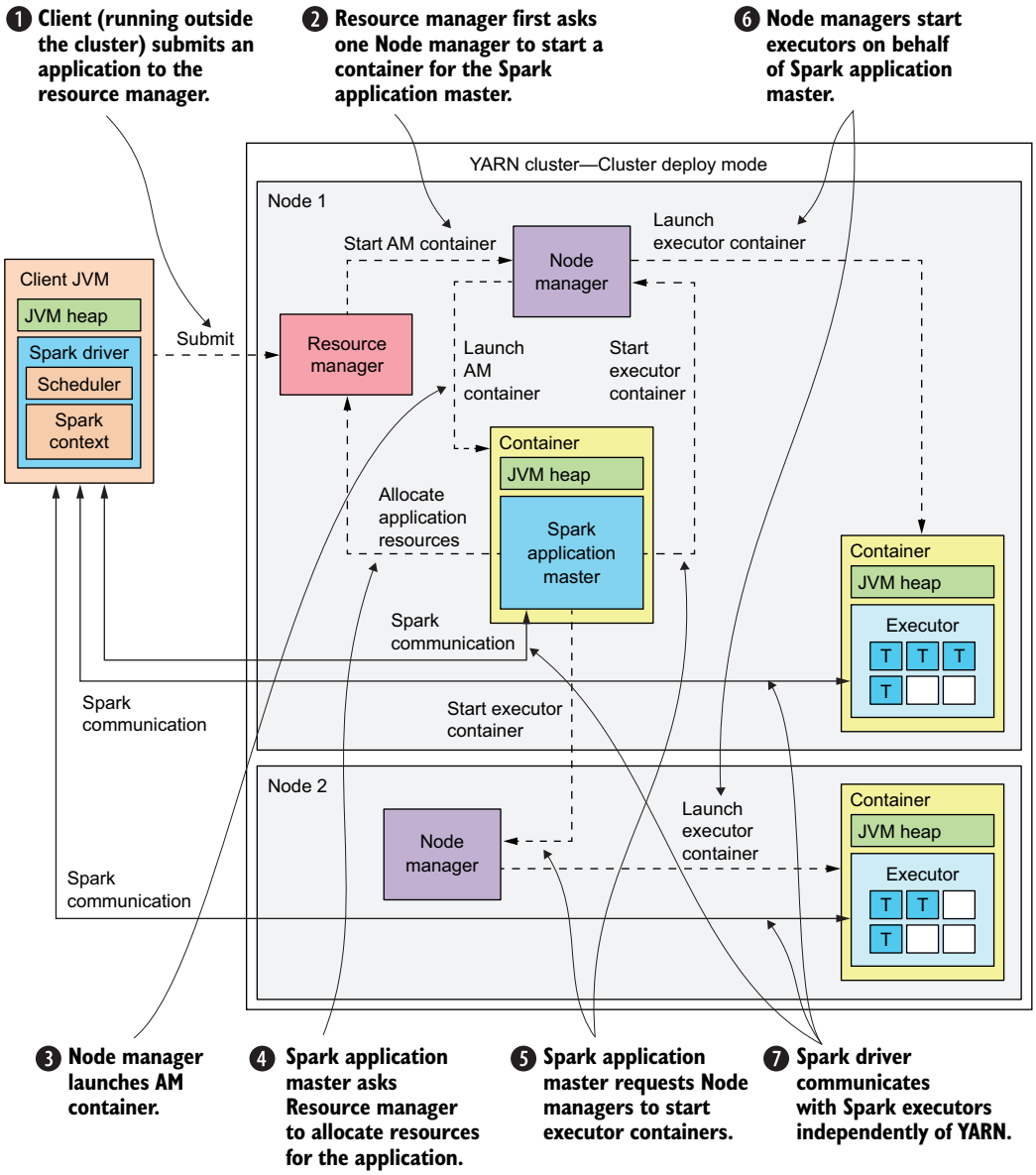

Figure 12.2 **Running spark in YARN client-deploy mode in a cluster with two nodes. The client submits an application, and the resource manager starts the container for the application master (Spark driver). The application master requests further containers for Spark executors. The Spark driver runs in the client JVM and directly communicates with the executors once the containers start.**

When you submit an application to YARN in cluster mode, your client process will stay alive and wait for the application to finish. Killing the client process won't stop the

application. If you've turned on information (INFO) message logging, your client process will display periodic messages like this one:

```
INFO Client: Application report for <application_id> (state: RUNNING)
```

You'll notice that application startup is somewhat slower on YARN than it is on a standalone cluster. This is because of the way YARN assigns resources: it first has to create a container for the application master, which then has to ask the resource manager to create containers for executors. This overhead isn't felt that much when running larger jobs.

STOPPING AN APPLICATION

YARN offers a way to stop a running application with the following command:

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

You can obtain the application ID from the `spark-submit` command's output if you enable logging of `INFO` messages for the `org.apache.spark` package. Otherwise, you can find the ID on the YARN web UI (see section 12.1.5). You can kill an application regardless of whether the application runs in client or cluster mode.

12.1.5 Configuring Spark on YARN

To run Spark on YARN, you only need to have Spark installed on the client node (the node running `spark-submit` or `spark-shell`, or if your application instantiates a `SparkContext`). The Spark assembly JAR and all configuration options are transferred automatically to the appropriate YARN containers.

The Spark distribution package needs to be built with YARN support. You can download a prebuilt version from the Spark official website (https://spark.apache.org/downloads.html) or build your own.

As you've probably noticed, the master connection URL for connecting to YARN contains no hostnames. Therefore, before submitting an application to a YARN cluster, you need to tell Spark where to find the YARN resource manager. This is done by setting one of the following two variables to point to the directory that contains the YARN configuration: `YARN_CONF_DIR` or `HADOOP_CONF_DIR`. At minimum, the specified directory needs to have at least one file, yarn-site.xml, with this configuration:

```
<?xml version="1.0"?>
<configuration>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>{RM_hostname}:{RM_port}</value>
    </property>
</configuration>
```

`RM_hostname` and `RM_port` are the hostname and port of your resource manager (the default port is 8050). For other configuration options that you can specify in yarn-site.xml, please see the official Hadoop documentation (http://mng.bz/zB92).

If you need to access HDFS, the directory specified by `YARN_CONF_DIR` or `HADOOP_CONF_DIR` should also contain the core-site.xml file with the parameter `fs.default`

.name set to a value similar to this: hdfs://yourhostname:9000. This client-side configuration will be distributed to all Spark executors in the YARN cluster.

SPECIFYING A YARN QUEUE

As we discussed in section 12.1.3, when using capacity or fair schedulers, YARN applications' resources are allocated by specifying a queue. You set the queue name Spark will use with the --queue command-line parameter, the spark.yarn.queue configuration parameter, or the SPARK_YARN_QUEUE environment variable. If you don't specify a queue name, Spark will use the default name (default).

SHARING THE SPARK ASSEMBLY JAR

When submitting Spark applications to a YARN cluster, JAR files containing Spark classes (all the JARs from Spark installation's jars folder) need to be transferred to the containers on remote nodes. This upload can take some time because these files can take up more than 150 MB. You can shorten this time by uploading the JARs to a specific folder on all executor machines manually and set spark.yarn.jars configuration parameter to point to that folder. Or you can make it point to a central folder on HDFS.

There is a third option. You can put the JAR files in an archive and set the spark.yarn.archive parameter to point to the archive (in a folder on each node or in an HDFS folder).

This way, Spark will be able to access the JARs from each container when needed, instead of uploading the JARs from the client each time it runs.

MODIFYING THE CLASSPATH AND SHARING FILES

Most of the things we said in the previous chapter about specifying extra classpath entries and files for a standalone cluster also apply to YARN. You can use a SPARK_CLASSPATH variable, the --jars command-line parameter, or spark.driver.extraClassPath (and others in spark.*.extra[Class|Library]Path). For more details, see the official documentation at http://mng.bz/75KO.

There are a few differences, though. An additional command-line parameter, --archives, lets you specify an archive name that will be transferred to worker machines and extracted into the working directory of each executor. Additionally, for each file or archive specified with --archives and --files, you can add a reference filename so you can access it from your program running on executors. For example, you can submit your application with these parameters:

```
--files /path/to/myfile.txt#fileName.txt
```

Then, you can access the file myfile.txt from your program by using the name fileName.txt. This is specific to running on YARN.

12.1.6 Configuring resources for Spark jobs

YARN schedules CPU and memory resources. A few specifics that you should be aware of are described in this section. For example, the default number of executor cores should be changed in most circumstances (as described shortly). Also, to take advantage

of Spark's memory management, it's important to configure it properly, especially on YARN. Several important, YARN-specific parameters are mentioned here as well.

SPECIFYING CPU RESOURCES FOR AN APPLICATION

The default setup when running on YARN is to have two executors and one core per executor. This usually isn't enough. To change this, use the following command-line options when submitting an application to YARN:

- `--num-executors`—Changes the number of executors
- `--executor-cores`—Changes the number of cores per executor

SPARK MEMORY MANAGEMENT WHEN RUNNING ON YARN

Just as on a standalone cluster, driver memory can be set with the `--driver-memory` command-line parameter, a `spark.driver.memory` configuration parameter, or the `SPARK_DRIVER_MEMORY` environment variable. For executors, the situation is a bit different. Similar to a standalone cluster, `spark.executor.memory` determines the executors' heap size (same as the `SPARK_EXECUTOR_MEMORY` environment variable or the `--executor-memory` command-line parameter). An additional parameter, `spark.executor.memoryOverhead`, determines additional memory beyond the Java heap that will be available to YARN containers running Spark executors. This memory is necessary for the JVM process itself. If your executor uses more memory than `spark.executor.memory + spark.executor.memoryOverhead`, YARN will shut down the container, and your jobs will repeatedly fail.

> **TIP** Failing to set `spark.executor.memoryOverhead` to a sufficiently high value can lead to problems that are hard to diagnose. Make sure to specify at least 1024 MB.

The memory layout of Spark's executors when running on YARN is shown in figure 12.3. It shows memory overhead along with the sections of the Java heap we described in section 10.2.4: storage memory (whose size is determined by `spark.storage.memoryFraction`), shuffle memory (size determined by `spark.shuffle.memoryFraction`), and the rest of the heap used for Java objects.

When your application is running in YARN cluster mode, `spark.yarn.driver.memoryOverhead` determines the memory overhead of the driver's container. `spark.yarn.am.memoryOverhead` determines the memory overhead of the application master in

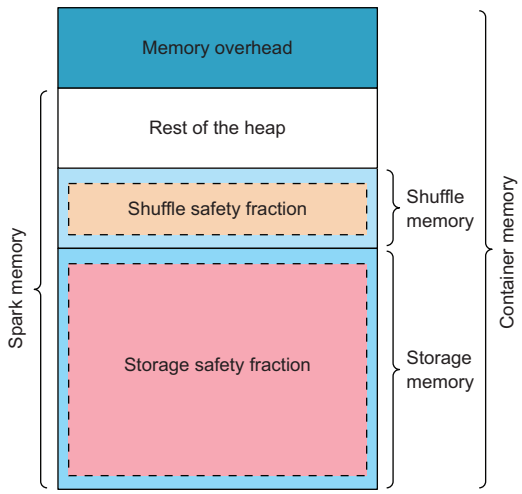

Figure 12.3 Components of Spark's executors' memory on YARN

client mode. Furthermore, a few YARN parameters (specified in yarn-site.xml) influence memory allocation:

- `yarn.scheduler.maximum-allocation-mb`—Determines the upper memory limit of YARN containers. The resource manager won't allow allocation of larger amounts of memory. The default value is 8192 MB.
- `yarn.scheduler.minimum-allocation-mb`—Determines the minimum amount of memory the resource manager can allocate. The resource manager allocates memory only in multiples of this parameter. The default value is 1024 MB.
- `yarn.nodemanager.resource.memory-mb`—Determines the maximum amount of memory YARN can use on a node overall. The default value is 8192 MB.

`yarn.nodemanager.resource.memory-mb` should be set to the amount of memory available on a node, minus the memory needed for the OS. `yarn.scheduler.maximum-allocation-mb` should be set to the same value. Because YARN will round up all allocation requests to multiples of `yarn.scheduler.minimum-allocation-mb`, that parameter should be set to a value small enough to not waste memory unnecessarily (for example, 256 MB).

CONFIGURING EXECUTOR RESOURCES PROGRAMMATICALLY

Executor resources can also be specified when creating Spark context objects programmatically with the `spark.executor.cores`, `spark.executor.instances`, and `spark.executor.memory` parameters. But if your application is running in YARN cluster mode, the driver is running in its own container, started in parallel with executor containers, so these parameters will have no effect. In that case, it's best to set these parameters in the spark-defaults.conf file or on the command line.

12.1.7 YARN UI

Similar to a Spark standalone cluster, YARN also provides a web interface for monitoring the cluster state. By default, it starts on port 8088 on the machine where the resource manager is running. You can see the state and various metrics of your nodes, the current capacity usage of the cluster, local and remote log files, and the status of finished and currently running applications. Figure 12.4 shows a sample YARN UI starting page with a list of applications.

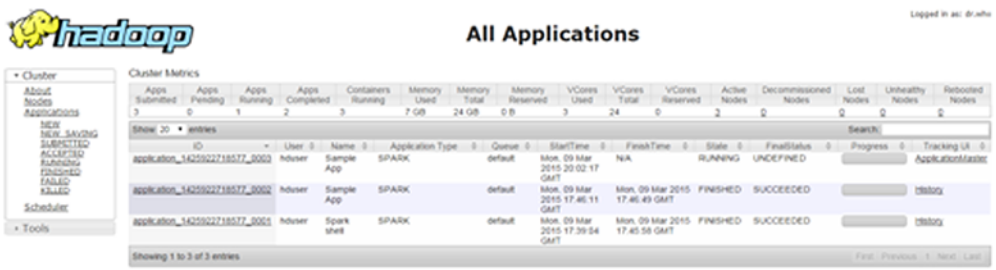

Figure 12.4 YARN UI: the All Applications page shows a list of running and finished applications

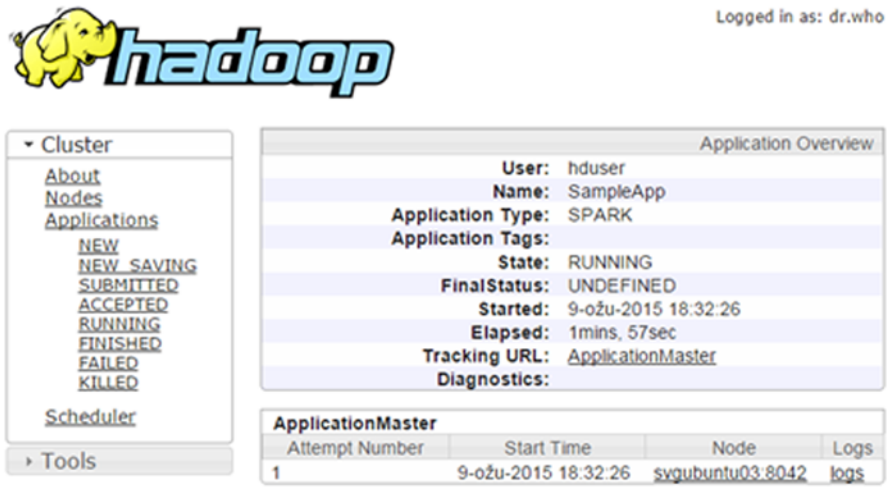

Figure 12.5 YARN UI: The Application Overview page

From this page, you can click an application's ID, which takes you to the Application Overview page (figure 12.5). There you can examine the application's name, status, and running time, and access its log files. The node at which the application master is running is also displayed.

ACCESSING THE SPARK WEB UI FROM THE YARN UI

The so-called *tracking UI* (or *tracking URL*) is available from the YARN UI Applications page and from the Application Overview page. The tracking URL takes you to the Spark web UI if the application is still running. We described the Spark web UI in chapter 10.

> **TIP** If you get the error "Connection refused" in your browser when trying to access the Spark web UI through the Tracking UI link, you should set the YARN configuration parameter `yarn.resourcemanager.hostname` (in yarn-site.xml) to the exact value of your YARN hostname.

When the application finishes, you'll be taken to the YARN Application Overview page (figure 12.4). If you have the Spark History Server running, the tracking URL for the finished application will point to the Spark History Server.

SPARK HISTORY SERVER AND YARN

If you enabled event logging and started the Spark History Server (as described in chapter 10), you'll be able to access the Spark web UI of a finished Spark application, just like when running on a Spark standalone cluster. But if you're running your application in YARN cluster mode, the driver can be run on any node in the cluster. In order for the Spark History Server to see the event log files, don't use a local directory as an event log directory; put it on HDFS.

12.1.8 Finding logs on YARN

By default, YARN stores your application's logs locally on the machines where the containers are running. The directory for storing log files is determined by the parameter `yarn.nodeamanager.log-dirs` (set in yarn-site.xml), which defaults to <Hadoop installation directory>/logs/userlogs. To view the log files, you need to look in this directory on each container's machine. The directory contains a subdirectory for each application.

You can also find the log files through the YARN web UI. The application master's logs are available from the Application Overview page. To view logs from other containers, you need to find the appropriate node by clicking Nodes and then clicking the node name in the list of nodes. Then go to its list of containers, where you can access logs for each of them.

USING LOG AGGREGATION

Another option on YARN is to enable the *log aggregation* feature by setting the `yarn.log-aggregation-enable` parameter in yarn-site.xml to `true`. After an application finishes, its log files will be transferred to a directory on HDFS as specified by the parameter `yarn.nodemanager.remote-app-log-dir`, which defaults to /tmp/logs. Under this directory, a hierarchy of subdirectories is created, first by the current user's username and then by the application ID. The final application aggregate log directory contains one file per node on which it executed.

You can view these aggregate log files by using the `yarn logs` command (only after the application has finished executing) and specifying the application ID:

```
$ yarn logs -applicationId <application_id>
```

As we already said, you can obtain the application ID from the `spark-submit` command's output if logging of `INFO` messages for the `org.apache.spark` package is enabled. Otherwise, you can find it on the YARN web UI (see section 12.1.5).

You can view a single container's logs by specifying the container's ID. To do so, you also need to specify the hostname and port of the node on which the container is executing. You can find this information on the Nodes page of the YARN web UI:

```
$ yarn logs -applicationId <application_id> -containerId <container_id> \
-nodeAddress <node hostname>:<node port>
```

You can then use shell utilities to further filter and grep the logs.

CONFIGURING THE LOGGING LEVEL

If you want to use an application-specific log4j configuration, you need to upload your log4j.properties file using the `--files` option while submitting the application. Alternatively, you can specify the location of the log4j.properties file (which should already be present on the node machines) using the `-Dlog4j.configuration` parameter in the `spark.executor.extraJavaOptions` option. For example, from the command line, you can do it like this:

```
$ spark-submit --master yarn-client --conf spark.executor.extraJavaOptions=
-Dlog4j.configuration=file:/usr/local/conf/log4j.properties" ...
```

12.1.9 Security considerations

Hadoop provides the means for authorizing access to resources (HDFS files, for example) to certain users, but it has no means of user authentication. Hadoop instead relies on Kerberos, a widely used and robust security framework. Hadoop allows user access or not, depending on Kerberos-provided identity and access control lists in the Hadoop configuration. If Kerberos is enabled in a Hadoop cluster (in other words, the cluster is *Kerberized*), only Kerberos-authenticated users can access it. YARN knows how to handle Kerberos authentication information and pass it on to HDFS. The Spark standalone and Mesos cluster managers don't have this functionality. You'll need to use YARN to run Spark if you need to access Kerberized HDFS.

To submit jobs to a Kerberized YARN cluster, you need a Kerberos *service principal name* (in the form `username/host@KERBEROS_REALM_NAME`; the *host* part is optional) and a *keytab* file. The service principal name serves as your Kerberos user name. Your keytab file contains pairs of user names and encryption keys used for encrypting Kerberos authentication messages. Your service principal name and keytab file are typically provided by your Kerberos administrator.

Before submitting a job to a Kerberized YARN cluster, you need to authenticate with a Kerberos server using the `kinit` command (on Linux systems):

```
$ kinit -kt <your_keytab_file> <your_service_principal>
```

Then submit your job as usual.

12.1.10 Dynamic resource allocation

As you may recall from the previous chapters, Spark applications obtain executors from the cluster manager and use them until they finish executing. The same executors are used for several jobs of the same application, and executors' resources remain allocated even though they may be idle between jobs. This enables tasks to reuse data from a previous job's tasks that ran on the same executors. For example, spark-shell may be idle for a long time while the user is away from their computer, but the executors it allocated remain, holding the cluster's resources.

Dynamic allocation is Spark's remedy for this situation, enabling applications to release executors temporarily so that other applications can use the allocated resources. This option has been available since Spark 1.2, but only for the YARN cluster manager. Since Spark 1.5, it's also available on Mesos and standalone clusters.

USING DYNAMIC ALLOCATION

You enable dynamic allocation by setting the `spark.dynamicAllocation.enabled` parameter to `true`. You should also enable Spark's shuffle service, which is used to serve executors' shuffle files even after the executors are no longer available. If an executor's shuffle files are requested and the executor isn't available while the service isn't enabled, shuffle files will need to be recalculated, which wastes resources. Therefore, you should always enable the shuffle service when enabling dynamic allocation.

To enable the shuffle service on YARN, you need to add spark-<version>-shuffle.jar (available from the lib directory of the Spark distribution) to the classpath of all node managers in the cluster. To do this, put the file in the share/hadoop/yarn/lib folder of your Hadoop installation and then add or edit the following properties in the yarn-site.xml file:

- Set the property `yarn.nodemanager.aux-services` to the value `"mapreduce_shuffle,spark_shuffle"` (basically, add `spark_shuffle` to the string).
- set the property `yarn.nodemanager.aux-services.spark_shuffle.class` to `org.apache.spark.network.yarn.YarnShuffleService`.

This will start the service in each node manager in your cluster. To tell Spark that it should use the service, you need to set the `spark.shuffle.service.enabled` Spark parameter to `true`.

When dynamic allocation is configured and running, Spark will measure the time during which there are pending tasks to be executed. If this period exceeds the interval specified by the parameter `spark.dynamicAllocation.schedulerBacklogTimeout` (in seconds), Spark will request executors from the resource manager. It will continue to request them every `spark.dynamicAllocation.sustainedSchedulerBacklogTimeout` seconds if there are pending tasks. Every time Spark requests new executors, the number of executors requested increases exponentially so that it can respond to the demand quickly enough—but not too quickly, in case the application only needs a couple of them. The parameter `spark.dynamicAllocation.executorIdleTimeout` specifies the number of seconds an executor needs to remain idle before it's removed.

You can control the number of executors with these parameters:

- `spark.dynamicAllocation.minExecutors`—Minimum number of executors for your application
- `spark.dynamicAllocation.maxExecutors`—Maximum number of executors for your application
- `spark.dynamicAllocation.initialExecutors`—Initial number of executors for your application

12.2 Running Spark on Mesos

Mesos is the last cluster manager supported by Spark that we'll talk about, but it's certainly not the least. We mentioned that the Spark project was originally started in order to demonstrate the usefulness of Mesos. Apple's Siri application runs on Mesos, as well as applications at eBay, Netflix, Twitter, Uber, and many other companies.

Mesos provides a distributed systems kernel and serves commodity cluster resources to applications, just like a Linux kernel manages a single computer's resources and serves them to applications running on a single machine. Mesos supports applications written in Java, C, C++, and Python. With version 0.20, Mesos can

run Docker containers, which are packages containing all the libraries and configurations that an application needs in order to run. With Docker support, you can run Mesos on virtually any application that can run in a Docker container. Since Spark 1.4, Spark can run on Mesos in Docker containers, too.

A few points where Mesos can use some improvement are security and support to run stateful applications (ones that use persistent storage, such as databases). With the current version (1.0.1), it isn't advisable to run stateful applications on Mesos. The community is working on supporting these use cases, too.[2] Also, Kerberos-based authentication isn't supported yet (https://issues.apache.org/jira/browse/MESOS-907). Applications, however, can provide authentication using Simple Authentication and Security Layer (SASL, a widely used authentication and data security framework), and intra-cluster communication is secured with Secure Sockets Layer (SSL). Dynamic allocation (explained in section 12.1.10), previously reserved for YARN only, is available on Mesos with Spark 1.5.

To begin our exploration of running Spark on Mesos, we'll first examine the Mesos architecture more closely. Then we'll see how running Spark on Mesos is different than running it on YARN and in Spark standalone clusters.

12.2.1 Mesos architecture

It's simpler to compare the Mesos architecture to a Spark standalone cluster than to YARN. Mesos's basic components—*masters*, *slaves*, and *applications* (or *frameworks*, in Mesos terms)—should be familiar to you from chapter 11. As is the case with a Spark standalone cluster, a Mesos master schedules slave resources among applications that want to use them. Slaves launch the application's executors, which execute tasks.

So far, so good. Mesos is more powerful than a Spark standalone cluster and differs from it in several important points. First, it can schedule other types of applications besides Spark (Java, Scala, C, C++, and Python applications). It's also capable of scheduling disk space, network ports, and even custom resources (not just CPU and memory). And instead of applications *demanding* resources from the cluster (from its master), a Mesos cluster *offers* resources to applications, which they can accept or refuse.

Frameworks running on Mesos (such as Spark applications) consist of two components: a *scheduler* and an *executor*. The scheduler accepts or rejects resources offered by the Mesos master and automatically starts Mesos executors on slaves. Mesos executors run tasks as requested by the frameworks' schedulers.

Figure 12.6 shows Spark running on a two-node Mesos cluster in client-deploy and coarse-grained modes. You'll learn what *coarse-grained* means in an instant.

[2] For more information, see https://issues.apache.org/jira/browse/MESOS-1554.

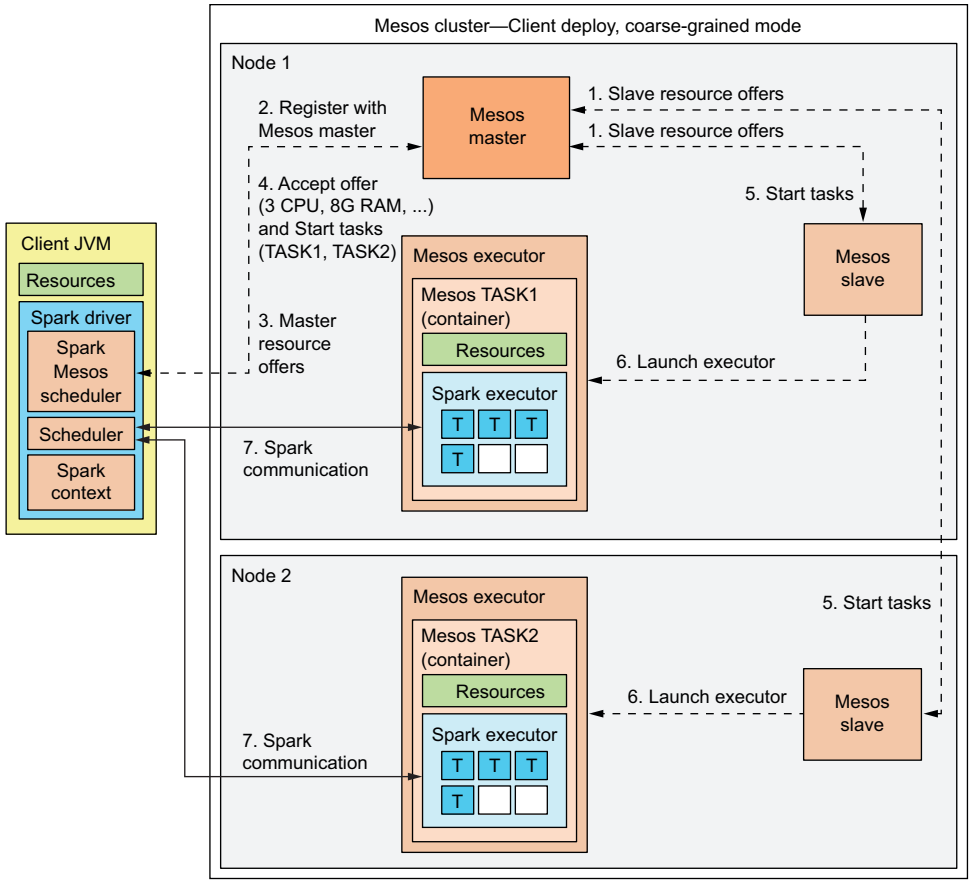

Figure 12.6 Spark running on a two-node Mesos cluster in client-deploy and coarse-grained modes

Let's explain the communication steps shown on the figure:

1. Mesos slaves offer their resources to the master.
2. Spark's Mesos-specific scheduler, running in a driver (the Spark `submit` command, for example) registers with the Mesos master.
3. The Mesos master in turn offers the available resources to the Spark Mesos scheduler (this happens continually: by default, every second while the framework is alive).
4. Spark's Mesos scheduler accepts some of the resources and sends a list of resources, along with a list of tasks it wants to run using the resources, to the Mesos master.
5. The master asks the slaves to start the tasks with the requested resources.

6 Slaves launch executors (in this case, Mesos's Command Executors), which launch Spark executors (using the provided command) in task containers.
7 Spark executors connect to the Spark driver and freely communicate with it, executing Spark tasks as usual.

You can see that the figure is similar to the ones for YARN and a Spark standalone cluster. The main difference is that the scheduling is backward: applications don't demand, but accept resources offered by the cluster manager.

FINE-GRAINED AND COARSE-GRAINED SPARK MODES

As we said, figure 12.6 shows Spark running in coarse-grained mode. This means Spark starts one Spark executor per Mesos slave. These executors stay alive during the entire lifetime of the Spark application and run tasks in much the same way as they do on YARN and a Spark standalone cluster.

Contrast this with fine-grained mode, shown on figure 12.7. Spark's fine-grained mode is available only on Mesos (it isn't available on other cluster types).

In fine-grained mode, one Spark executor—and, hence, one Mesos task—is started per Spark task. This means much more communication, data serialization, and setting up of Spark executor processes will need to be done, compared to coarse-grained mode. Consequently, jobs are likely to be slower in fine-grained mode than in coarse-grained mode.

The rationale behind fine-grained mode is to use cluster resources more flexibly so that other frameworks running on the cluster can get a chance to use some of the resources a Spark application may not currently need. It's used mainly for batch or streaming jobs that have long-running tasks, because in those cases, the slowdown due to management of Spark executors is negligible.

One additional detail visible in figure 12.7 is the Spark Mesos executor. This is a custom Mesos executor used only in Spark fine-grained mode.

Fine-grained mode is the default option, so if you try Mesos and see that it's much slower than YARN or a Spark standalone cluster, first switch to coarse-grained mode by setting the `spark.mesos.coarse` parameter to `true` (configuring Spark was described in chapter 10) and try your job again. Chances are, it will be much faster.

MESOS CONTAINERS

Tasks in Mesos are executed in *containers*, shown in figures 12.6 and 12.7, whose purpose is to isolate resources between processes (tasks) on the same slave so that tasks don't interfere with each other. The two basic types of containers in Mesos are *Linux cgroups containers* (default containers) and *Docker containers*.

- *cgroups (control groups)*—A feature of the Linux kernel that limits and isolates processes' resource usage. A control group is a collection of processes to which a set of resource limitations (CPU, memory, network, disk) is applied.
- Docker containers—Similar to VMs. In addition to limiting resources, as cgroups containers do, Docker containers provide the required system libraries. This is the crucial difference.

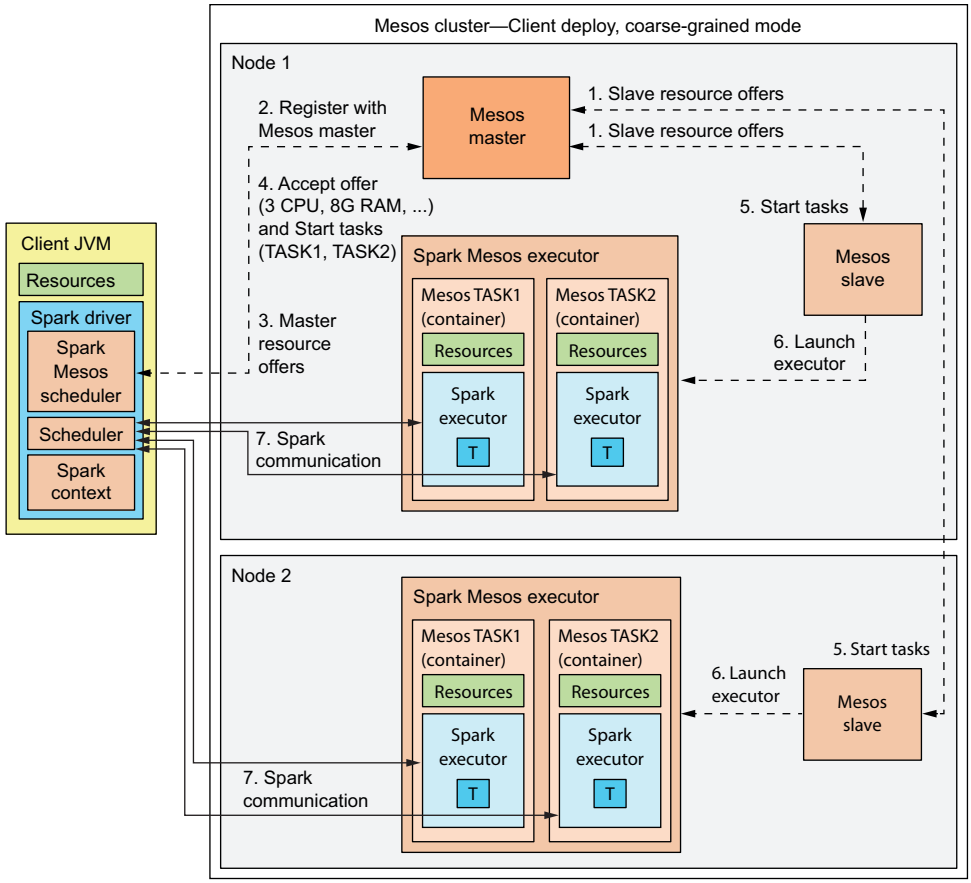

Figure 12.7 Spark running on a two-node Mesos cluster in client-deploy and fine-grained modes. Each Spark executor runs only one task.

We'll say more about using Docker in Mesos in section 12.2.6.

MASTER HIGH-AVAILABILITY

Similar to Spark standalone clusters, you can set up Mesos to use several master processes. Master processes use ZooKeeper to elect a leader among themselves. If the leader goes down, the standby masters will elect a new leader, again using ZooKeeper.

12.2.2 Installing and configuring Mesos

The officially recommended way to install Mesos is to build it from source code.[3] But if you're lucky enough to be running a Linux version supported by Mesosphere, a quicker and simpler way is to install Mesos from the Mesosphere package repository.[4]

[3] For more information, see the "Getting Started" guide at http://mesos.apache.org/gettingstarted.
[4] For more information, see https://mesosphere.com/downloads.

Here we show the steps for installing Mesos from Mesosphere on Ubuntu. If you need help installing Mesos on other platforms or more information about installing and configuring Mesos in general, we recommend *Mesos in Action* by Roger Ignazio (Manning, 2016).

You first need to set up the repository

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
$ echo "deb http://repos.mesosphere.io/ubuntu trusty main" | \
    sudo tee /etc/apt/sources.list.d/mesosphere.list
```

and then install the package:

```
$ sudo apt-get install mesos
```

On the master node, you'll also need to install Zookeeper:

```
$ sudo apt-get install zookeeper
```

BASIC CONFIGURATION

You're (almost) good to go. The only thing left is to tell the slaves where to find the master. Master and slaves look in the file /etc/mesos/zk to find ZooKeeper's master address (which you should set up and start before starting Mesos). Slaves always ask ZooKeeper for the master's address.

Once you edit /etc/mesos/zk, you have a fully working Mesos cluster. If you want to further customize your Mesos configuration, you can use these locations:

```
/etc/mesos
/etc/mesos-master
/etc/mesos-slave
/etc/default/mesos
/etc/default/mesos-master
/etc/default/mesos-slave
```

The list of all configuration options is available at the official documentation page (http://mesos.apache.org/documentation/latest/configuration). You can also obtain it by running the `mesos-master --help` and `mesos-slave --help` commands. Environment variables are specified in the /etc/default/mesos* files just listed. Command-line parameters can also be specified on the command line and in the /etc/mesos/* directories. Place each parameter in a separate file, where the name of the file matches the parameter name and the contents of the file contain the parameter value.

STARTING MESOS

You run Mesos by starting the corresponding service. To start the master, you should use

```
$ sudo service mesos-master start
```

Use this for slaves:

```
$ sudo service mesos-slave start
```

These commands automatically pick up the configurations from the configuration files previously described. You can verify that the services are running by accessing the Mesos web UI at port 5050 (the default port).

12.2.3 Mesos web UI

When you start a Mesos master, it automatically starts a web UI interface. Figure 12.8 shows the main page.

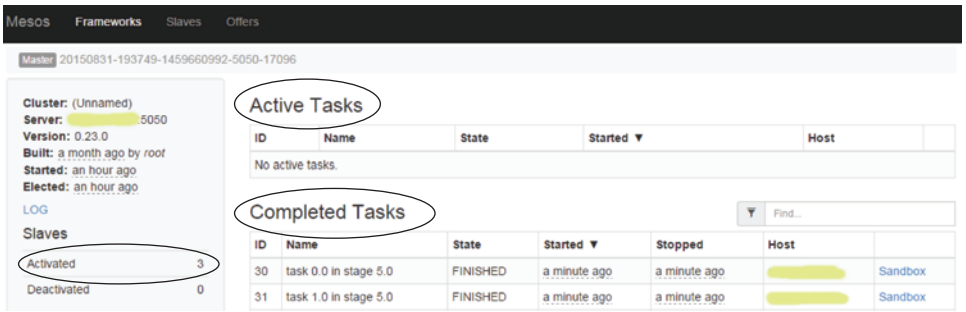

Figure 12.8 The main Mesos web UI page shows active and completed tasks and basic information about the cluster.

On the main page, you'll find a list of active and completed tasks, information about the master and slaves, and an overview of the cluster's CPU and memory (figure 12.9). The list of active and terminated frameworks is available on the Frameworks page (figure 12.10).

You can click one of the frameworks to examine a list of its active and terminated tasks. The Slaves page (figure 12.11) shows a list of registered slaves and their resources. This is where

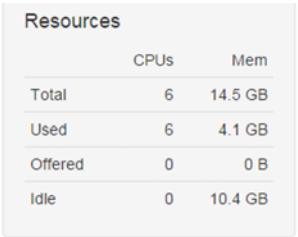

Figure 12.9 Overview of the state of the cluster's CPU and memory resources on the main Mesos web UI page

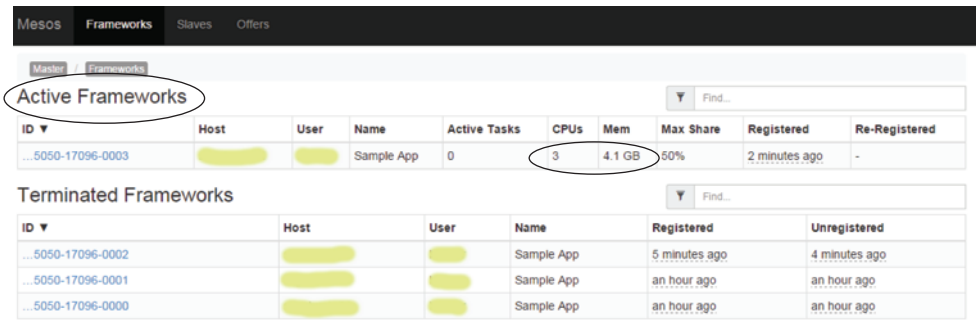

Figure 12.10 The Mesos Frameworks page shows active and terminated frameworks, with an overview of their resources.

you can check whether all of your slaves have registered with the master and are available. You can click one of the slaves to see a list of previous or currently running frameworks. If you click one of the frameworks, you'll also see a list of its running and completed executors on that slave.

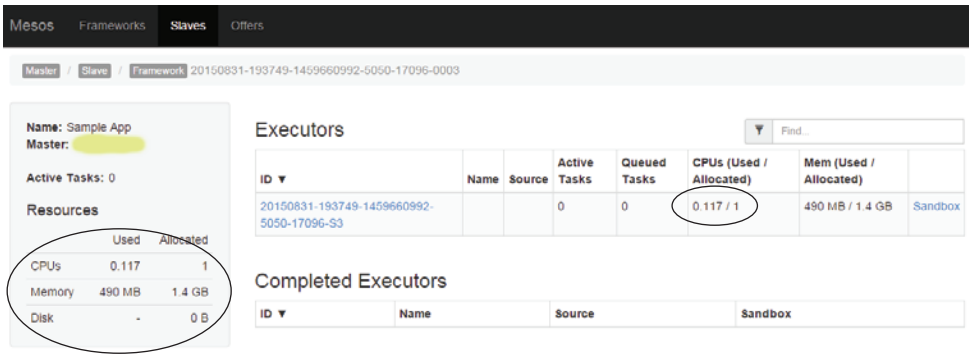

Figure 12.11 A list of a framework's executors on a slave. The executor's environment is available from the Sandbox link.

From this screen, you can access an executor's environment by clicking the Sandbox link. Figure 12.12 shows the resulting page.

The Sandbox page lets you see your executor's environment: application files and its log files. When you click one of the log files, a separate window opens with a live view of your application's logs, which is automatically updated as new lines become available. This is useful for debugging your application.

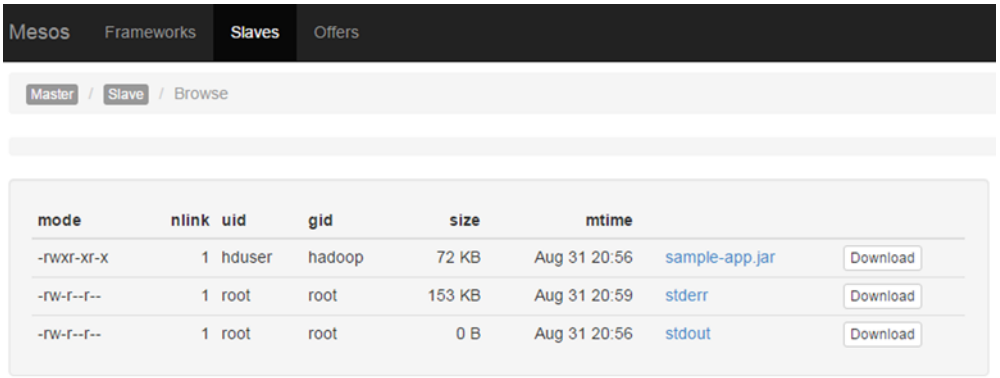

Figure 12.12 An executor's Sandbox page shows the executor's working folder. You can also download application files and system out and system error log files.

12.2.4 Mesos resource scheduling

The fine-grained and coarse-grained scheduling described in section 12.2.1 are Spark's scheduling policies when running on Mesos. Mesos itself knows nothing about them.

Resource scheduling decisions in Mesos are made on two levels: by the *resource-allocation module* running in the Mesos master, and by the *framework's scheduler*, which we mentioned previously. A resource-allocation module decides which resources to offer to which frameworks and in which order. As we mentioned, Mesos can schedule memory, CPU, disk, network ports, and even custom resources. It seeks to satisfy the needs of all frameworks while fairly distributing resources among them.

Different types of workloads and different frameworks require different allocation policies, because some are long-running and some are short-running, some mostly use memory, some CPU, and so forth. That's why a resource-allocation module allows for use of plug-ins that can change resource-allocation decisions.

By default, Mesos uses the Dominant Resource Fairness (DRF) algorithm, which is appropriate for the majority of use cases. It tracks each framework's resources and determines the *dominant resource* for each framework, which is the resource type the framework uses the most. Then it calculates the framework's *dominant share*, which is the share of all cluster resources of the dominant resource type used by the framework. If a framework is using 1 CPU and 6 GB of RAM in a cluster of 20 CPUs and 36 GB of RAM, then it's using 1/20 of all CPUs and 1/6 of all RAM; thus its dominant resource is RAM, and its dominant share is 1/6.

DRF allocates the newest resource to the framework with the lowest dominant share. The framework can accept or reject the offer. If it currently has no work to do, or if the offer doesn't contain enough resources for its needs, the framework can reject the offer. This can be useful if the framework needs a specific locality level for its tasks. It can wait until a resource offer with an acceptable locality level becomes available. If a framework is holding on to some resources for too long, Mesos can revoke the tasks taking up the resources.

ROLES, WEIGHTS, AND RESOURCE ALLOCATIONS

Roles and weights let you prioritize frameworks so that some get more resource offers than others. *Roles* are names attached to typical groups of users or frameworks, and *weights* are numbers attached to roles that specify priorities the roles will have when resource-allocation decisions are made.

You can initialize a master with a list of acceptable roles with the `--roles` option (for example, `--roles=dev,test,stage,prod`) or with a list of weights per role (for example, `--weights='dev=30,test=50,stage=80,prod=100'`). As we explained in the previous section, you can place these values in the /etc/mesos-master/roles and /etc/mesos-master/weights files, respectively. In this example, the `prod` role will get twice as many resource offers as the `test` role.

Furthermore, resource allocations can be used on slaves to reserve some resources for a particular role. When starting a slave, you can specify the `--resources` parameter with the contents formatted as follows: `'resource_name1(role_name1):value1;'resource_name2(role_name2):value2'`. For example, `--resources='cpu(prod):3;mem(prod):2048'` will reserve 3 CPUs and 2 GB of RAM exclusively for frameworks in the `prod` role.

Frameworks can specify a role when registering with a master. To tell Spark which role to use when registering Mesos, you can use the `spark.mesos.role` parameter (which is available with Spark version 1.5).

MESOS ATTRIBUTES AND SPARK CONSTRAINTS

Mesos also lets you specify custom attributes per slave with the `--attributes` parameter. You can use attributes to specify the type and version of the slave's operating system or the rack the slave is running on. Mesos supports attributes of the following types: float, integer, range, set, and text. For more information, see the official documentation (http://mesos.apache.org/documentation/attributes-resources/). Once specified, the attributes are sent with each resource offer.

Beginning with Spark 1.5, you can specify attribute constraints using the `spark.mesos.constraints` parameter. This instructs Spark to accept only those offers whose attributes match the specified constraints. You specify the constraints as key-value pairs, separated by semicolons (`;`). Keys and values themselves are separated by colons (`:`); and if the value is composed of several values, they're separated by commas (`,`) (for example, `os:ubuntu,redhat;zone:EU1,EU2`).

12.2.5 Submitting Spark applications to Mesos

The master URL for submitting Spark applications to Mesos starts with mesos:// and needs to specify the master's hostname and port:

```
$ spark-submit --master mesos://<master_hostname>:<master_port>
```

The default Mesos master port is 5050. If you have several masters running, synchronized through Zookeeper, you can instruct Spark to ask Zookeeper for the currently elected leader by specifying a master URL in this format:

```
mesos://zk://<zookeeper_hostname>:<zookeeper_port>
```

The ZooKeeper port defaults to 2181.

MAKING SPARK AVAILABLE TO SLAVES

Mesos's slaves need to know where to find the Spark classes to start your tasks. If you have Spark installed on the slave machines in the same location as on the driver, then there's nothing special you need to do to make Spark available on the slave machines. Mesos's slaves will automatically pick up Spark from the same location.

If Spark is installed on the slave machines, but in a different location, then you can specify this location with the Spark parameter `spark.mesos.executor.home`. If you

don't have Spark installed on the slave machines, then you need to upload Spark's binary package to a location accessible by slaves. This can be an NFS shared filesystem or a location on HDFS or Amazon S3. You can get Spark's binary package either by downloading it from Spark's official download page or by building Spark yourself and packaging the distribution with the `make-distribution.sh` command. Then you need to set this location as the value of the parameter `spark.executor.uri` *and* as the value of the environment variable `SPARK_EXECUTOR_URI`.

RUNNING IN CLUSTER MODE

Mesos's cluster mode is available with Spark 1.4. A new component was added to Spark for this purpose: `MesosClusterDispatcher`. It's a separate Mesos framework used only for submitting Spark drivers to Mesos.

You start `MesosClusterDispatcher` with the script `start-mesos-dispatcher.sh` from Spark's sbin directory and then pass the master URL to it. Start the cluster dispatcher with ZooKeeper's version of the master's URL. Otherwise, you may have problems submitting jobs (for example, `submit` could hang). The complete `start` command looks like this:

```
$ start-mesos-dispatcher.sh --master mesos://zk://<bind_address>:2181/mesos
```

The dispatcher process will start listening at port 7077. You can then submit applications to the dispatcher process instead of the Mesos master. You also need to specify cluster-deploy mode when submitting applications:

```
$ spark-submit --deploy-mode cluster --master \
   mesos://<dispatcher_hostname>:7077 ...
```

If everything goes OK, the driver's Mesos task will be visible in the Mesos UI, and you'll be able to see which slave it's running. You can use that information to access the driver's Spark web UI.

OTHER CONFIGURATION OPTIONS

A few more parameters are available for configuring how Spark runs on Mesos, in addition to the `spark.mesos.coarse`, `spark.mesos.role`, and `spark.mesos.constraints` parameters that we mentioned before. They're shown in table 12.2.

Table 12.2 Additional configuration parameters for running Spark on Mesos

Parameter	Default	Description
`spark.cores.max`	All available cores	Limits the number of tasks your cluster can run in parallel. It's available on other cluster managers, too.
`spark.mesos.mesosExecutor.cores`	1	Tells Spark how many cores to reserve per Mesos executor, in addition to the resources it takes for its tasks. The value can be a decimal number.

Table 12.2 Additional configuration parameters for running Spark on Mesos *(continued)*

Parameter	Default	Description
`spark.mesos.extra.cores`	0	Specifies the extra number of cores to reserve per task in coarse-grained mode.
`spark.mesos.executor.memoryOverhead`	10% of `spark.executor.memory`	Specifies the amount of extra memory to reserve per executor. This is similar to the memory overhead parameter on YARN.

12.2.6 Running Spark with Docker

As we said earlier, Mesos uses Linux cgroups as the default "containerizer" for the tasks it's running, but it can also use Docker. Docker lets you package an application along with all of its library and configuration requirements. The name *Docker* comes from an analogy to freight containers, which all adhere to the same specifications and sizes so they can be handled and transported the same way regardless of their contents. The same principle runs true for Docker containers: they can run in different environments (different OSes with different versions of Java, Python, or other libraries), but they behave the same way on all of them because they bring their own libraries with them. So, you only need to set up your machines to run Docker containers. Docker containers bring everything needed for whichever application they contain.

You can use Docker, Mesos, and Spark in several interesting ways and combinations. You can run Mesos in Docker containers, or you can run Spark in Docker containers. You can have Mesos run Spark and other applications in Docker containers, or you can do both—run Mesos in Docker containers and have it run other Docker containers. EBay, for example, uses Mesos to run Jenkins servers for continuous delivery in the company's development department (http://mng.bz/MLtO).

The benefit of running Docker containers on Mesos is that Docker and Mesos provide two layers of abstraction between your application and the infrastructure it's running on, so you can write your application for a specific environment (contained in Docker) and distribute it to hundreds and thousands of different machines. We'll first show you how to install and configure Docker and then how to use it to run Spark jobs on Mesos.

INSTALLING DOCKER

Installing Docker on Ubuntu is simple because the installation script is available online at https://get.docker.com/.[5] You only need to pass it to your shell:

```
$ curl -sSL https://get.docker.com/ | sh
```

You can verify your installation with this command:

```
$ sudo docker run hello-world
```

[5] For other environments, see the official installation documentation: https://docs.docker.com/installation.

To enable other users to run the `docker` command without having to use `sudo` every time, create a user group named `docker` and add your user to the group:

```
$ sudo addgroup docker
$ sudo usermod -aG docker <your_username>
```

After logging out and logging back in, you should be able to run `docker` without using the `sudo` command.

USING DOCKER

Docker uses *images* to run containers. Images are comparable to templates, and containers are comparable to concrete instances of the images. You can *build* new images from *Dockerfiles*, which describe the image contents, and you can *pull* images from Docker Hub, an online repository of public Docker images. If you visit Docker Hub (https://hub.docker.com) and search for *mesos* or *spark*, you'll see that dozens of images are available for you to use. In this section, you'll build a Docker image based on the mesosphere/mesos image available from Docker Hub.

When you have *built* or *pulled* an image to your local machine, it's available locally, and you can *run* it. If you try to run an image that doesn't exist locally, Docker pulls it automatically from the Docker Hub. To list all the images available locally, you can use the `docker images` command. To see the running containers, use the `docker ps` command. For the complete command reference, use the `docker --help` command.

You can also run a command in a container in interactive mode by using the `-i` and `-t` flags, and by specifying the command as an additional argument. For example, this command starts a `bash` shell in a container of an image called `spark-image`:

```
$ docker run -it spark-image bash
```

BUILDING A SPARK DOCKER IMAGE

This is the contents of a Dockerfile for building a Docker image that can be used to run Spark on Mesos:

```
FROM mesosphere/mesos:0.20.1
RUN apt-get update && \
    apt-get install -y python libnss3 openjdk-7-jre-headless curl
RUN mkdir /opt/spark && \
    curl http://www-us.apache.org/dist/spark/
➥ spark-2.0.0/spark-2.0.0-bin-hadoop2.7.tgz | tar -xzC /opt
ENV SPARK_HOME /opt/spark-2.0.0-bin-hadoop2.7
ENV MESOS_NATIVE_JAVA_LIBRARY /usr/local/lib/libmesos.so
```

Copy those lines into a file called Dockerfile in a folder of your choosing. Then, position yourself in the folder and build the image using this command:

```
$ docker build -t spark-mesos .
```

This command instructs Docker to build an image called *spark-mesos* using the Docker file in the current directory. You can verify that your image is available with the `docker images` command. You need to do this *on all the slave machines* in your Mesos cluster.

PREPARING MESOS TO USE DOCKER

Before using Docker in Mesos, the Docker containerizer needs to be enabled. You need to execute the following two commands on each of the slave machines in your Mesos cluster:

```
$ echo 'docker,mesos' > /etc/mesos-slave/containerizers
$ echo '5mins' > /etc/mesos-slave/executor_registration_timeout
```

Then restart your slaves:

```
$ sudo service mesos-slave restart
```

Mesos should now be ready to run Spark executors in Docker images.

RUNNING SPARK TASKS IN DOCKER IMAGES

Before running Spark tasks in your newly built Docker image, you need to set a couple of configuration parameters. First, you need to tell Spark the name of the image by specifying it in the `spark.mesos.executor.docker.image` parameter (in your spark-defaults.conf file). The image you built has Spark installed in the folder /opt/spark-2.0.0-bin-hadoop2.7, which is probably different than your Spark installation location. So, you'll need to set the parameter `spark.mesos.executor.home` to the image's Spark installation location. And you'll need to tell Spark executors where to find Mesos's system library. You can do this with the parameter `spark.executorEnv.MESOS_NATIVE_JAVA_LIBRARY`.

Your spark-defaults.conf file should now contain these lines:

```
spark.mesos.executor.docker.image           spark-mesos
spark.mesos.executor.home            /opt/spark-2.0.0-bin-hadoop2.7
spark.executorEnv.MESOS_NATIVE_JAVA_LIBRARY  /usr/local/lib/libmesos.so
```

We'll use the `SparkPi` example to demonstrate submitting an application to Docker, but you can use your own (for example, the one built in chapter 3). Position yourself in your Spark home directory, and issue the following command (the name of the JAR file could be different, depending on your Spark version):

```
$ spark-submit --master mesos://zk://<your_hostname>:2181/mesos \
   --class org.apache.spark.examples.SparkPi \
   examples/jars/spark-examples_2.11-2.0.0.jar
```

If everything goes well, you should see the message *Pi is roughly 3.13972* in your console output. To verify that Mesos really used a Docker container to run Spark tasks, you can use the Mesos UI to find your framework. Click the Sandbox link next to one

of framework's completed tasks, and then open its standard output log file. It should contain the following line:

```
Registered docker executor on <slave's_hostname>
```

FURTHER DOCKER CONFIGURATION

If you need to access the slave's filesystem from within your Docker image, Docker lets you mount the host's folders to the folders in your image with the `-v` flag. You can instruct Spark to do this for you when launching Docker executors by specifying the parameter `spark.mesos.executor.docker.volumes` containing a comma-separated list of volume (folder) mappings in the following format:

```
[host_path:]container_path[:ro|:rw]
```

The host path is optional if it's the same as the container path.

Docker also lets you connect certain network ports on your image to the ones on the slave's host. You can specify these port mappings with the parameter `spark.mesos.executor.docker.portmaps` in this format:

```
host_port:container_port[:tcp|:udp]
```

This way, you'll be able to access the container through ports on the host.

12.3 Summary

- YARN is the new generation of Hadoop's MapReduce execution engine and can run MapReduce, Spark, and other types of programs.
- YARN consists of a resource manager and several node managers.
- Applications on YARN run in containers and provide their application masters.
- YARN supports three different schedulers: FIFO, capacity, and fair.
- Spark on YARN runs in yarn-cluster and yarn-client modes.
- YARN kills containers that use more memory than allowed, so tuning `spark.executor.memoryOverhead` is important.
- YARN provides log aggregation for easy log inspection.
- YARN was the first cluster manager to support dynamic allocation.
- YARN is the only cluster manager on which Spark can access HDFS secured with Kerberos.
- Mesos can also run different types of applications (you can even run YARN on Mesos), but unlike YARN, it's capable of scheduling disk, network, and even custom resources.
- Mesos consists of masters, slaves, frameworks, and executors.
- Spark on Mesos can run in fine-grained or coarse-grained mode.
- Mesos provides resource isolation for its tasks through containers, implemented with Linux cgroups or Docker.

- Mesos's resource scheduling operates on two levels: by a framework's scheduler and by Mesos's resource-allocation module.
- Mesos allows framework prioritization through roles, weights, and resource allocations.
- You can use Spark's contraints to accept resource requests from only some of the slaves in the cluster.
- Spark on Mesos supports both client and cluster modes. Cluster mode is implemented with Spark's Mesos dispatcher.
- You can run Spark's executors on Mesos from Docker images.