- **randomwriter**: A map/reduce program that writes 10 GB of random data per node.
- **secondarysort**: An example defining a secondary sort to the reduce.
- **sort:** A map/reduce program that sorts the data written by the random writer.
- **sudoku:** A Sudoku solver.
- **teragen:** Generate data for the terasort.
- **terasort:** Run the terasort.
- **teravalidate:** Check the results of the terasort.
- **wordcount:** A map/reduce program that counts the words in the input files.
- **wordmean:** A map/reduce program that counts the average length of the words in the input files.
- **wordmedian:** A map/reduce program that counts the median length of the words in the input files.
- **wordstandarddeviation:** A map/reduce program that counts the standard deviation of the length of the words in the input files.

Some of the examples require files to be copied to or from HDFS. For those unfamiliar with basic HDFS operation, an HDFS quick start is provided in Appendix F. If you were able to complete the preceding steps, you should now have a fully functioning Apache Hadoop YARN system running in pseudo-distributed mode.

## Wrap-up

With a working installation of YARN, the concepts, examples, and applications found in this book can be explored further without the need for a large production cluster. Keep in mind that many aspects of the configuration were simplified for this single-machine installation. In particular, a single workstation/server install does not have a true parallel HDFS or parallel MapReduce environment. Additional production installation scenarios are provided in Chapter 5, "Installing Apache Hadoop YARN."

*This page intentionally left blank*

# 3

# Apache Hadoop YARN Core Concepts

The new Apache Hadoop YARN resource manager is introduced in this chapter. In addition to allowing non-MapReduce tasks to work within a Hadoop installation, YARN ("Yet Another Resource Negotiator") provides several other advantages over the previous version of Hadoop, including better scalability, cluster utilization, and user agility.

YARN also brings with it several new services that separate it from the standard Hadoop MapReduce model. A new ResourceManager acting purely as a resource scheduler is the sole arbitrator of cluster resources. User applications, including MapReduce jobs, ask for specific resource requests via the new ApplicationMaster component, which in turn negotiates with the ResourceManager to create an application container within the cluster.

By incorporating MapReduce as a YARN framework, YARN also provides full backward compatibility with existing MapReduce tasks and applications.

## Beyond MapReduce

The Apache Hadoop ecosystem continues to grow beyond the simple MapReduce job. Although MapReduce remains at the core of many Hadoop 1.0 tasks, the introduction of YARN has expanded the capability of a Hadoop environment to move beyond the basic MapReduce process.

The basic structure of Hadoop with Apache Hadoop MapReduce version 1 (MRv1) can be seen in Figure 3.1. The two core services, Hadoop File System (HDFS) and MapReduce, form the basis for almost all Hadoop functionality. All other components are built around these services and must use MapReduce to run Hadoop jobs.

Apache Hadoop provides a basis for large-scale MapReduce processing and has spawned a Big Data ecosystem of tools, applications, and vendors. While MapReduce methods enable users to focus on the problem at hand rather than the underlying

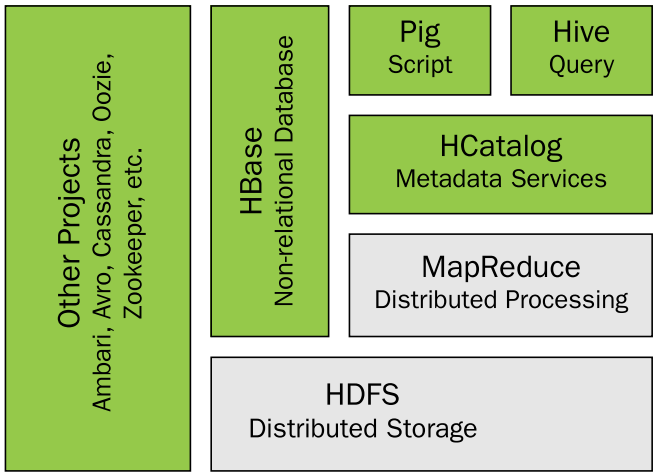



**Figure 3.1** The Hadoop 1.0 ecosystem. MapReduce and HDFS are the core components, while other components are built around the core.

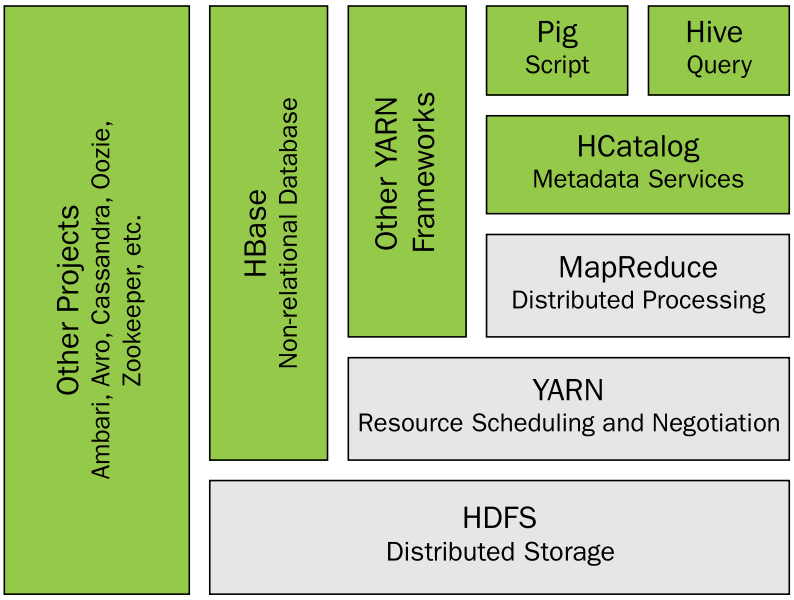



**Figure 3.2** YARN adds a more general interface to run non-MapReduce jobs within the Hadoop framework

processing mechanism, they do limit some of the problem domains that can run in the Hadoop framework.

To address these needs, the YARN project was started by the Apache Hadoop community to give Hadoop the ability to run non-MapReduce jobs within the Hadoop framework. YARN provides a generic resource management framework for implementing distributed applications. Starting with Apache Hadoop version 2.0, MapReduce has undergone a complete overhaul; it is now architected as an application on YARN to be called MapReduce version 2 (MRv2). YARN provides both full compatibility with existing MapReduce applications and new support for virtually any distributed application. Figure 3.2 illustrates how YARN fits into the new Hadoop ecosystem.

The introduction of YARN does not change the ability of Hadoop to run MapReduce jobs. It does, however, position MapReduce as merely one of the application frameworks within Hadoop, which works the same way as it did in MRv1. The new capability offered by YARN is the use of new non-MapReduce frameworks that add many new features to the Hadoop ecosystem.

### The MapReduce Paradigm

The MapReduce processing model consists of two separate steps. The first step is an embarrassingly parallel map phase, in which input data is split into discrete chunks that can be processed independently. The second step is a reduce phase, in which the output of the map phase is aggregated to produce the desired result. The simple, and fairly restricted, nature of the programming model lends itself to very efficient and extremely large-scale implementations across thousands of low-cost commodity servers (or nodes). When MapReduce is paired with a distributed file system such as Apache Hadoop HDFS, which can provide very high aggregate I/O bandwidth across a large cluster of commodity servers, the economics of the system become extremely compelling—a key factor in the popularity of Hadoop.

One of the keys to Hadoop performance is the *lack of data motion*, such that compute tasks are moved to the servers on which the data reside and not the other way around (i.e., large data movement to compute servers is minimized or eliminated). Specifically, the MapReduce tasks can be scheduled on the same physical nodes on which data reside in HDFS, which exposes the underlying storage layout across the cluster. This design significantly reduces the network I/O patterns and keeps most of the I/O on the local disk or on a neighboring server within the same server rack.

## Apache Hadoop MapReduce

To understand the new YARN process flow, it is helpful to review the original Apache Hadoop MapReduce design. As part of the Apache Software Foundation, Apache Hadoop MapReduce has evolved and improved as an open-source project. This project is an implementation of the MapReduce programming paradigm described previously. The Apache Hadoop MapReduce project itself can be broken down into the following major facets:

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

This separation of concerns has significant benefits, particularly for end-users where they can completely focus on their application via the API and let the combination of the MapReduce run-time and the framework deal with the complex details such as resource management, fault tolerance, and scheduling.

The current Apache Hadoop MapReduce system is composed of several high-level elements, as shown in Figure 3.3. The master process is the JobTracker, which serves as the clearinghouse for all MapReduce jobs on in the cluster. Each node has a TaskTracker process that manages tasks on the individual node. The TaskTrackers communicate with and are controlled by the JobTracker.

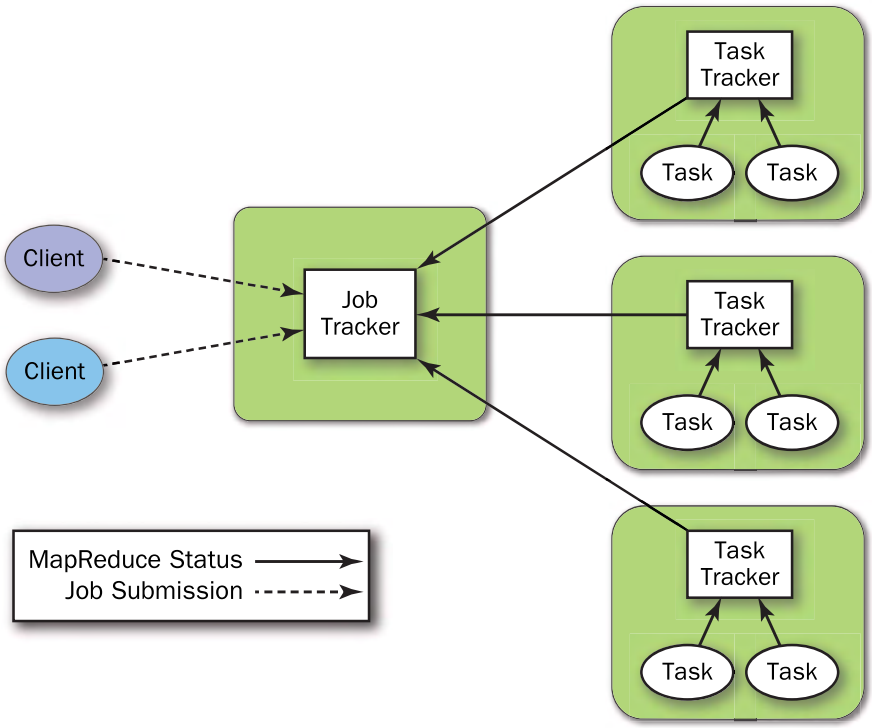



Figure 3.3 Current Hadoop MapReduce control elements

The JobTracker is responsible for *resource management* (managing the worker server nodes—that is, the TaskTrackers), *tracking resource consumption/availability,* and *job life-cycle management* (e.g., scheduling individual tasks of the job, tracking progress, providing fault tolerance for tasks).

The TaskTracker has simple responsibilities—launch/tear down tasks on orders from the JobTracker and provide task status information to the JobTracker periodically.

As described in Chapter 1, the Apache Hadoop MapReduce framework has exhibited some growing pains. In particular, with regard to the JobTracker, several aspects, including scalability, cluster utilization, ability of users to control upgrades to the stack (i.e., user agility), and support for workloads other than MapReduce itself, have been identified as desirable features.

## The Need for Non-MapReduce Workloads

MapReduce is great for many applications, but not everything; other programming models better serve requirements such as graph processing (e.g., Google Pregel/Apache Giraph) and iterative modeling using the Message Passing Interface (MPI). As is often the case, much of the enterprise data is already available in Hadoop HDFS, and having multiple paths for processing is critical and a clear necessity. Furthermore, given that MapReduce is essentially batch oriented, support for real-time and near-real-time processing has become an important issue for the user base. A more robust computing environment within Hadoop will enable organizations to see an increased return on their Hadoop investments by lowering operational costs for administrators, reducing the need to move data between Hadoop HDFS and other storage systems, and providing other such efficiencies.

## Addressing Scalability

The processing power available in data centers continues to increase rapidly. As an example, consider the additional hardware capability offered by a commodity server over a three-year period:

- 2009: 8 cores, 16 GB of RAM, 4 × 1 TB disk
- 2012: 16+ cores, 72 GB of RAM, 12 × 3 TB of disk

These new servers are often available at the same price point as those of previous generations. In general, servers are twice as capable today as they were two to three years ago—on every dimension. Apache Hadoop MapReduce is known to scale to production deployments of approximately 5000 server nodes of 2009 vintage. Thus, for the same price, the number of CPU cores, amount of RAM, and local storage available to the user will put continued pressure on the scalability of new Apache Hadoop installations.

### Improved Utilization

In the current system, the JobTracker views the cluster as composed of nodes (managed by individual TaskTrackers) with distinct map slots and reduce slots, which are not *fungible*. As discussed in Chapter 1, utilization issues occur because maps slots might be "full," while reduce slots are empty (and vice versa). Improving this situation is necessary to ensure the entire system could be used to its maximum capacity for high utilization and applying resources when needed.

### User Agility

In real-world deployments, Hadoop is very commonly offered as a shared, multitenant system. As a result, changes to the Hadoop software stack affect a large cross-section of the enterprise, if not the entire enterprise. Against that backdrop, users are very keen to control upgrades to the software stack, as such upgrades have a direct impact on their applications. Thus, allowing multiple, if limited, number of versions of the MapReduce framework is critical for Hadoop.

## Apache Hadoop YARN

The fundamental idea of YARN is to split the two major responsibilities of the JobTracker—that is, resource management and job scheduling/monitoring—into separate daemons: a global ResourceManager and a per-application ApplicationMaster (AM). The ResourceManager and per-node slave, the NodeManager (NM), form the new, and *generic,* operating system for managing applications in a distributed manner.

The ResourceManager is the ultimate authority that arbitrates division of resources among all the applications in the system. The per-application ApplicationMaster is, in effect, a *framework-specific* entity and is tasked with negotiating for resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the component tasks.

The ResourceManager has a pluggable scheduler component, which is responsible for allocating resources to the various running applications subject to the familiar constraints of capacity, queues, and other factors. The Scheduler is a *pure scheduler* in the sense that it performs no monitoring or tracking of status for the application, offering no guarantees on restarting tasks that are not carried out due to either application failure or hardware failure. The scheduler performs its scheduling function based on the *resource requirements* of an application by using the abstract notion of a *resource container*, which incorporates resource dimensions such as memory, CPU, disk, and network.

The NodeManager is the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage (CPU, memory, disk, network), and reporting the same to the ResourceManager.

The per-application ApplicationMaster is responsible for negotiating appropriate resource containers from the Scheduler, tracking their status, and monitoring their progress. From the system perspective, the ApplicationMaster runs as a normal *container.* An architectural view of YARN is provided in Figure 3.4.

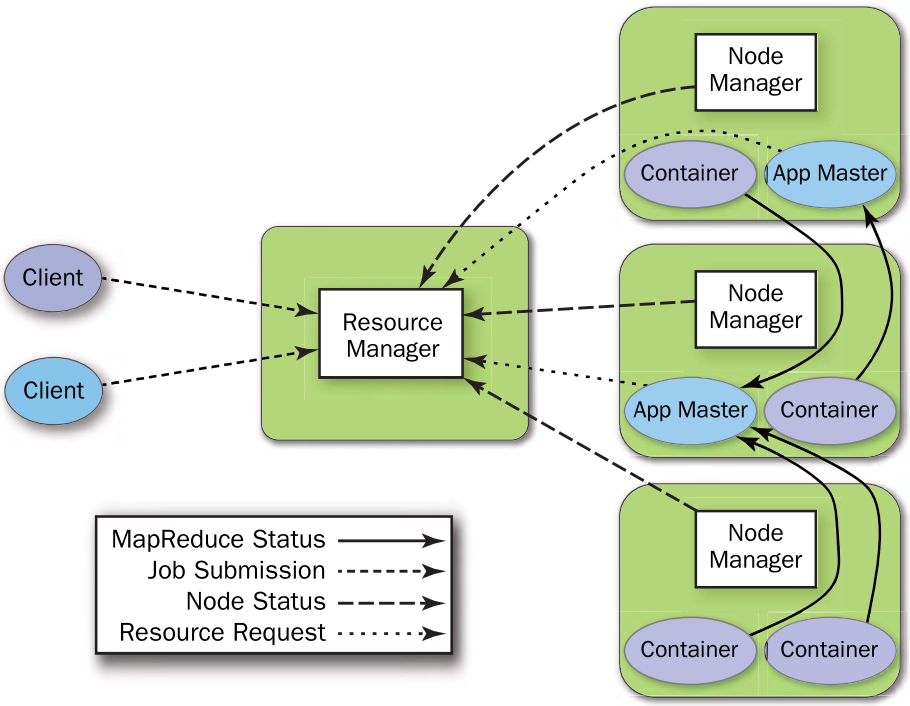



**Figure 3.4** New YARN control elements

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

# YARN Components

By adding new functionality, YARN brings new components into the Apache Hadoop workflow. These components provide finer-grained control for the end-user and simultaneously offer more advanced capabilities to the Hadoop ecosystem.

## ResourceManager

As mentioned earlier, the YARN ResourceManager is primarily a *pure scheduler*. It is strictly limited to arbitrating requests for available resources in the system made by the competing applications. It optimizes for cluster utilization (i.e., keeps all resources in use all the time) against various constraints such as capacity guarantees, fairness,

and service level agreements (SLAs). To allow for different policy constraints, the ResourceManager has a *pluggable scheduler* that enables different algorithms such as those focusing on capacity and fair scheduling to be used as necessary.

## ApplicationMaster

An important new *concept* in YARN is the ApplicationMaster. The ApplicationMaster is, in effect, an *instance* of a *framework-specific library* and is responsible for negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the containers and their resource consumption. It has the responsibility of negotiating for appropriate resource containers from the ResourceManager, tracking their status, and monitoring progress.

The ApplicationMaster design enables YARN to offer the following important new features:

- Scale: The ApplicationMaster provides much of the job-oriented functionality of the JobTracker so that the entire system can scale more dramatically. Simulations have shown that jobs may scale to 10,000 node clusters composed of modern hardware without significant issue. As a pure scheduler, the ResourceManager does not, for example, have to provide fault tolerance for resources across the cluster. By shifting fault tolerance to the ApplicationMaster instance, control becomes local, rather than global. Furthermore, because an instance of an ApplicationMaster is made available per application, the ApplicationMaster itself is rarely a bottleneck in the cluster.
- Openness: Moving all application framework-specific code into the ApplicationMaster generalizes the system so that it can now support multiple frameworks such as MapReduce, MPI, and Graph Processing.

These features were the result of some key YARN design decisions:

- Move all complexity (to the extent possible) to the ApplicationMaster, while providing sufficient functionality to allow application framework authors sufficient flexibility and power.
- Because it is essentially user code, do not trust the ApplicationMaster(s). In other words, no ApplicationMaster is a privileged service.
- The YARN system (ResourceManager and NodeManager) has to protect itself from faulty or malicious ApplicationMaster(s) and resources granted to them at all costs.

In reality, every application has its own instance of an ApplicationMaster. However, it's completely feasible to implement an ApplicationMaster to manage a set of applications (e.g., ApplicationMaster for Pig or Hive to manage a set of MapReduce jobs). Furthermore, this concept has been stretched to manage long-running services, which manage their own applications (e.g., launching HBase in YARN via a special HBaseAppMaster).

## Resource Model

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

- Resource name (including hostname, rackname, and possibly complex network topologies)
- Amount of memory
- CPUs (number/type of cores)
- Eventually resources such as disk/network I/O, GPUs, and more

## ResourceRequests and Containers

YARN is designed to allow individual applications (via the ApplicationMaster) to utilize cluster resources in a shared, secure, and multitenant manner. It also remains aware of cluster topology so that it can efficiently schedule and optimize data access (i.e., reduce data motion for applications to the extent possible).

To meet those goals, the central Scheduler (in the ResourceManager) maintains extensive information about an application's resource needs, which allows it to make better scheduling decisions across all applications in the cluster. This leads us to the ResourceRequest and the resulting container.

Essentially, an application can ask for specific resource requests via the ApplicationMaster to satisfy its resource needs. The Scheduler responds to a resource request by allocating a container, which satisfies the requirements laid out by the ApplicationMaster in the initial ResourceRequest.

A ResourceRequest has the following form:

```
<resource-name, priority, resource-requirement, number-of-containers>
```

These components are described as follows:

- `resource-name` is either hostname, rackname where the resource is desired, or * to indicate no preference. Future plans may support even more complex topologies for virtual machines on a host, more complex networks, and other features.
- `priority` is intra-application priority for this request (not across multiple applications). This orders various ResourceRequests within a given application.
- `resource-requirement` is the required capabilities such as the amount of memory or CPU time (currently YARN supports only memory and CPU).
- `number-of-containers` is just a multiple of such containers. It limits the total number of containers as specified in the ResourceRequest.

Essentially, the container is the resource allocation, which is the successful result of the ResourceManager granting a specific ResourceRequest. A container grants rights to an application to use a specific amount of resources (e.g., memory, CPU) on a specific host.

The ApplicationMaster must take the container and present it to the NodeManager managing the host, on which the container was allocated, to use the resources for launching its tasks. For security reasons, the container allocation is verified, in the secure mode, to ensure that ApplicationMaster(s) cannot fake allocations in the cluster.

### Container Specification

While a container, as described previously, is merely the *right* to use a specified amount of resources on a specific machine (NodeManager) in the cluster, the ApplicationMaster must provide considerably more information to the NodeManager to actually *launch* the container. YARN allows applications to launch any process and, unlike existing Hadoop MapReduce, this ability isn't limited to Java applications.

The YARN container launch specification API is platform agnostic and contains the following elements:

- Command line to launch the process within the container
- Environment variables
- Local resources necessary on the machine prior to launch, such as jars, shared-objects, and auxiliary data files
- Security-related tokens

This design allows the ApplicationMaster to work with the NodeManager to launch containers ranging from simple shell scripts to C/Java/Python processes on UNIX/Windows to full-fledged virtual machines.

## Wrap-up

The release of Apache Hadoop YARN provides many new capabilities to the existing Hadoop Big Data ecosystem. While the scalable MapReduce paradigm has enabled previously intractable problems to be efficiently managed on large clustered systems, YARN provides a framework for managing both MapReduce and non-MapReduce tasks of greater size and complexity. YARN provides the framework to apply low-cost commodity hardware to virtually any Big Data problem.

# 4

# Functional Overview of YARN Components

YARN relies on three main components for all of its functionality. The first component is the ResourceManager (RM), which is the arbitrator of all cluster resources. It has two parts: a pluggable scheduler and an ApplicationManager that manages user jobs on the cluster. The second component is the per-node NodeManager (NM), which manages users' jobs and workflow on a given node. The central ResourceManager and the collection of NodeManagers create the unified computational infrastructure of the cluster. The third component is the ApplicationMaster, a user job life-cycle manager. The ApplicationMaster is where the user application resides. Together, these three components provide a very scalable, flexible, and efficient environment to run virtually any type of large-scale data processing jobs.

## Architecture Overview

The central ResourceManager runs as a standalone daemon on a dedicated machine and acts as the central authority for allocating resources to the various competing applications in the cluster. The ResourceManager has a central and global view of all cluster resources and, therefore, can provide fairness, capacity, and locality across all users. Depending on the application demand, scheduling priorities, and resource availability, the ResourceManager dynamically allocates resource containers to applications to run on particular nodes. A container is a logical bundle of resources (e.g., memory, cores) bound to a particular cluster node. To enforce and track such assignments, the ResourceManager interacts with a special system daemon running on each node called the NodeManager. Communications between the ResourceManager and NodeManagers are heartbeat based for scalability. NodeManagers are responsible for local monitoring of resource availability, fault reporting, and container life-cycle management (e.g., starting and killing jobs). The ResourceManager depends on the NodeManagers for its "global view" of the cluster.

User applications are submitted to the ResourceManager via a public protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed. Those applications that are accepted pass to the scheduler and are allowed to run. Once the scheduler has enough resources to satisfy the request, the application is moved from an accepted state to a running state. Aside from internal bookkeeping, this process involves allocating a container for the ApplicationMaster and spawning it on a node in the cluster. Often called "container 0," the ApplicationMaster does not get any additional resources at this point and must request and release additional containers.

The ApplicationMaster is the "master" user job that manages all life-cycle aspects, including dynamically increasing and decreasing resources consumption (i.e., containers), managing the flow of execution (e.g., in case of MapReduce jobs, running reducers against the output of maps), handling faults and computation skew, and performing other local optimizations. The ApplicationMaster is designed to run arbitrary user code that can be written in any programming language, as all communication with the ResourceManager and NodeManager is encoded using extensible network protocols (i.e., Google Protocol Buffers, http://code.google.com/p/protobuf/).

YARN makes few assumptions about the ApplicationMaster, although in practice it expects most jobs will use a higher-level programming framework. By delegating all these functions to ApplicationMasters, YARN's architecture gains a great deal of scalability, programming model flexibility, and improved user agility. For example, upgrading and testing a new MapReduce framework can be done independently of other running MapReduce frameworks.

Typically, an ApplicationMaster will need to harness the processing power of multiple servers to complete a job. To achieve this, the ApplicationMaster issues resource requests to the ResourceManager. The form of these requests includes specification of locality preferences (e.g., to accommodate HDFS use) and properties of the containers. The ResourceManager will attempt to satisfy the resource requests coming from each application according to availability and scheduling policies. When a resource is scheduled on behalf of an ApplicationMaster, the ResourceManager generates a lease for the resource, which is acquired by a subsequent ApplicationMaster heartbeat. A token-based security mechanism guarantees its authenticity when the ApplicationMaster presents the container lease to the NodeManager. In MapReduce, the code running in the container can be a map or a reduce task. Commonly, running containers will communicate with the ApplicationMaster through an application-specific protocol to report status and health information and to receive framework-specific commands. In this way, YARN provides a basic infrastructure for monitoring and life-cycle management of containers, while application-specific semantics are managed independently by each framework. This design is in sharp contrast to the original Hadoop version 1 design, in which scheduling was designed and integrated around managing only MapReduce tasks.

Figure 4.1 illustrates the relationship between the application and YARN components. The YARN components appear as the large outer boxes (ResourceManager and NodeManagers), and the two applications appear as smaller boxes (Containers), one

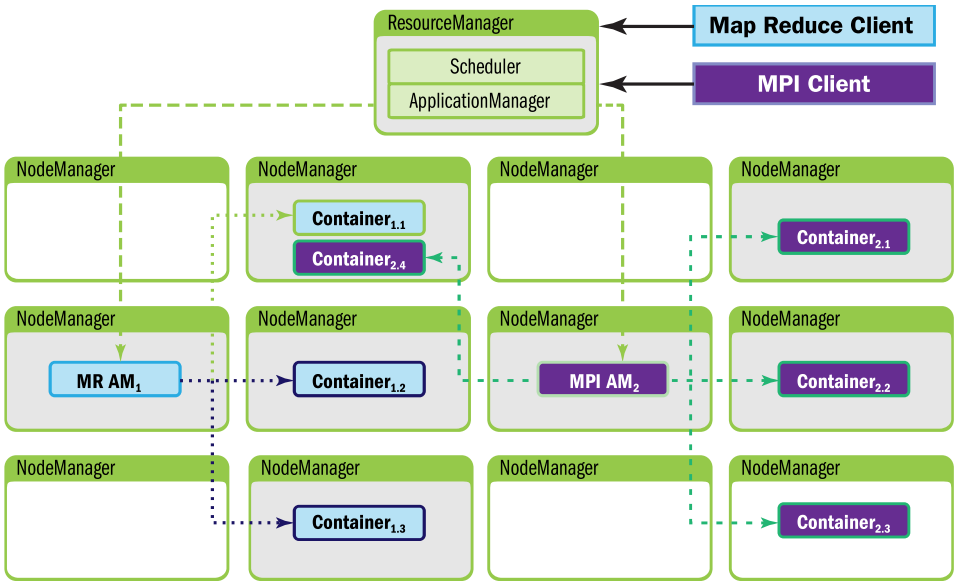



**Figure 4.1** YARN architecture with two clients (MapReduce and MPI). The client MPI $AM_2$ is running an MPI application and the client MR $AM_1$ is running a MapReduce application.

dark and one light. Each application uses a different ApplicationMaster; the darker client is running a Message Passing Interface (MPI) application and the lighter client is running a MapReduce application.

## ResourceManager

As previously described, the ResourceManager is the master that arbitrates all the available cluster resources, thereby helping manage the distributed applications running on the YARN system. It works together with the per-node NodeManagers and the per-application ApplicationMasters.

In YARN, the ResourceManager is primarily limited to scheduling—that is, it allocates available resources in the system among the competing applications but does not concern itself with per-application state management. The scheduler handles only an overall resource profile for each application, ignoring local optimizations and internal application flow. In fact, YARN completely departs from the static assignment of map and reduce slots because it treats the cluster as a resource pool. Because of this clear separation of responsibilities coupled with the modularity described previously, the ResourceManager is able to address the important design requirements of scalability and support for alternative programming paradigms.

In contrast to many other workflow schedulers, the ResourceManager also has the ability to symmetrically request back resources from a running application. This situation typically happens when cluster resources become scarce and the scheduler decides to reclaim some (but not all) of the resources that were given to an application.

In YARN, ResourceRequests can be strict or negotiable. This feature provides ApplicationMasters with a great deal of flexibility on how to fulfill the reclamation requests—for example, by picking containers to reclaim that are less crucial for the computation, by checkpointing the state of a task, or by migrating the computation to other running containers. Overall, this scheme allows applications to preserve work, in contrast to platforms that kill containers to satisfy resource constraints. If the application is noncollaborative, the ResourceManager can, after waiting a certain amount of time, obtain the needed resources by instructing the NodeManagers to forcibly terminate containers.

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

In contrast to the Hadoop 1.0 JobTracker, it is important to mention the tasks for which the ResourceManager is not responsible. Other than tracking application execution flow and task fault tolerance, the ResourceManager will not provide access to the application status (servlet; now part of the ApplicationMaster) or track previously executed jobs, a responsibility that is now delegated to the JobHistoryService (a daemon running on a separated node). This is consistent with the view that the ResourceManager should handle only live resource scheduling, and helps YARN central components scale better than Hadoop 1.0 JobTracker.

# YARN Scheduling Components

YARN has a pluggable scheduling component. Depending on the use case and user needs, administrators may select either a simple FIFO (first in, first out), capacity, or fair share scheduler. The scheduler class is set in `yarn-default.xml`. Information about the currently running scheduler can be found by opening the ResourceManager web UI and selecting the Scheduler option under the Cluster menu on the left (e.g., http://your_cluster:8088/cluster/scheduler). The various scheduler options are described briefly in this section.

## FIFO Scheduler

The original scheduling algorithm that was integrated within the Hadoop version 1 JobTracker was called the *FIFO* scheduler, meaning "first in, first out." The FIFO scheduler is basically a simple "first come, first served" scheduler in which the JobTracker pulls jobs from a work queue, oldest job first. Typically, FIFO schedules have

no sense of job priority or scope. The FIFO schedule is practical for small workloads, but is feature-poor and can cause issues when large shared clusters are used.

## Capacity Scheduler

The Capacity scheduler is another pluggable scheduler for YARN that allows for multiple groups to securely share a large Hadoop cluster. Developed by the original Hadoop team at Yahoo!, the Capacity scheduler has successfully been running many of the largest Hadoop clusters.

To use the Capacity scheduler, an administrator configures one or more queues with a predetermined fraction of the total slot (or processor) capacity. This assignment guarantees a minimum amount of resources for each queue. Administrators can configure soft limits and optional hard limits on the capacity allocated to each queue. Each queue has strict ACLs (Access Control Lists) that control which users can submit applications to individual queues. Also, safeguards are in place to ensure that users cannot view or modify applications from other users.

The Capacity scheduler permits sharing a cluster while giving each user or group certain minimum capacity guarantees. These minimums are not given away in the absence of demand. Excess capacity is given to the most starved queues, as assessed by a measure of running or used capacity divided by the queue capacity. Thus, the fullest queues as defined by their initial minimum capacity guarantee get the most needed resources. Idle capacity can be assigned and provides elasticity for the users in a cost-effective manner.

Queue definitions and properties such as capacity and ACLs can be changed, at run time, by administrators in a secure manner to minimize disruption to users. Administrators can add additional queues at run time, but queues cannot be deleted at run time. In addition, administrators can stop queues at run time to ensure that while existing applications run to completion, no new applications can be submitted.

The Capacity scheduler currently supports memory-intensive applications, where an application can optionally specify higher memory resource requirements than the default. Using information from the NodeManagers, the Capacity scheduler can then place containers on the best-suited nodes.

The Capacity scheduler works best when the workloads are well known, which helps in assigning the minimum capacity. For this scheduler to work most effectively, each queue should be assigned a minimal capacity that is less than the maximal expected workload. Within each queue, multiple applications are scheduled using hierarchical FIFO queues similar to the approach used with the stand-alone FIFO scheduler. Capacity Scheduler is covered in more detail in Chapter 8, "Capacity Scheduler in YARN."

## Fair Scheduler

The Fair scheduler is a third pluggable scheduler for Hadoop that provides another way to share large clusters. Fair scheduling is a method of assigning resources to applications such that all applications get, on average, an equal share of resources over time.

> **Note**
>
> In Hadoop version 1, the Fair scheduler uses the term "pool" to refer to a queue. Starting with the YARN Fair scheduler, the term "queue" will be used instead of "pool." To provide backward compatibility with the original Fair scheduler, "queue" elements can be named as "pool" elements.

In the Fair scheduler model, every application belongs to a queue. YARN containers are given to the queue with the least amount of allocated resources. Within the queue, the application that has the fewest resources is assigned the container. By default, all users share a single queue, called "default." If an application specifically lists a queue in a container resource request, the request is submitted to that queue. It is also possible to configure the Fair scheduler to assign queues based on the user name included with the request. The Fair scheduler supports a number of features such as weights on queues (heavier queues get more containers), minimum shares, maximum shares, and FIFO policy within queues, but the basic idea is to share the resources as uniformly as possible.

Under the Fair scheduler, when a single application is running, that application may request the entire cluster (if needed). If additional applications are submitted, resources that are free are assigned "fairly" to the new applications so that each application gets roughly the same amount of resources. The Fair scheduler also applies the notion of preemption, whereby containers can be requested back from the ApplicationMaster. Depending on the configuration and application design, preemption and subsequent assignment can be either friendly or forceful.

In addition to providing fair sharing, the Fair scheduler allows guaranteed minimum shares to be assigned to queues, which is useful for ensuring that certain users, groups, or production applications always get sufficient resources. When a queue contains waiting applications, it gets at least its minimum share; in contrast, when the queue does not need its full guaranteed share, the excess is split between other running applications. To avoid a single user flooding the clusters with hundreds of jobs, the Fair scheduler can limit the number of running applications per user and per queue through the configurations file. Using this limit, user applications will wait in the queue until previously submitted jobs finish.

The YARN Fair scheduler allows containers to request variable amounts of memory and schedules based on those requirements. Support for other resource specifications, such as type of CPU, is under development. To prevent multiple smaller memory applications from starving a single large memory application, a "reserved container" has been introduced. If an application is given a container that it cannot use immediately due to a shortage of memory, it can reserve that container, and no other application can use it until the container is released. The reserved container will wait until other local containers are released and then use this additional capacity (i.e., extra

RAM) to complete the job. One reserved container is allowed per node, and each node may have only one reserved container. The total reserved memory amount is reported in the ResourceManager UI. A larger number means that it may take longer for new jobs to get space.

A new feature in the YARN Fair scheduler is support for hierarchical queues. Queues may now be nested inside other queues, with each queue splitting the resources allotted to it among its subqueues in a fair scheduling fashion. One use of hierarchical queues is to represent organizational boundaries and hierarchies. For example, Marketing and Engineering departments may now arrange a queue structure to reflect their own organization. A queue can also be divided into subqueues by job characteristics, such as short, medium, and long run times.

The Fair scheduler works best when there is a lot of variability between queues. Unlike with the Capacity scheduler, all jobs make progress rather than proceeding in a FIFO fashion in their respective queues.

## Containers

At the fundamental level, a container is a collection of physical resources such as RAM, CPU cores, and disks on a single node. There can be multiple containers on a single node (or a single large one). Every node in the system is considered to be composed of multiple containers of minimum size of memory (e.g., 512 MB or 1 GB) and CPU. The ApplicationMaster can request any container so as to occupy a multiple of the minimum size.

A container thus represents a resource (memory, CPU) on a single node in a given cluster. A container is supervised by the NodeManager and scheduled by the ResourceManager.

Each application starts out as an ApplicationMaster, which is itself a container (often referred to as container 0). Once started, the ApplicationMaster must negotiate with the ResourceManager for more containers. Container requests (and releases) can take place in a dynamic fashion at run time. For instance, a MapReduce job may request a certain amount of mapper containers; as they finish their tasks, it may release them and request more reducer containers to be started.

## NodeManager

The NodeManager is YARN's per-node "worker" agent, taking care of the individual compute nodes in a Hadoop cluster. Its duties include keeping up-to-date with the ResourceManager, overseeing application containers' life-cycle management, monitoring resource usage (memory, CPU) of individual containers, tracking node health, log management, and auxiliary services that may be exploited by different YARN applications.

On start-up, the NodeManager registers with the ResourceManager; it then sends heartbeats with its status and waits for instructions. Its primary goal is to manage application containers assigned to it by the ResourceManager.

YARN containers are described by a container launch context (CLC). This record includes a map of environment variables, dependencies stored in remotely accessible storage, security tokens, payloads for NodeManager services, and the command necessary to create the process. After validating the authenticity of the container lease, the NodeManager configures the environment for the container, including initializing its monitoring subsystem with the resource constraints' specified application. The NodeManager also kills containers as directed by the ResourceManager.

## ApplicationMaster

The ApplicationMaster is the process that coordinates an application's execution in the cluster. Each application has its own unique ApplicationMaster, which is tasked with negotiating resources (containers) from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks. In the YARN design, MapReduce is just one application framework; this design permits building and deploying distributed applications using other frameworks. For example, YARN ships with a Distributed-Shell application that allows a shell script to be run on multiple nodes on the YARN cluster.

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

## YARN Resource Model

In earlier Hadoop versions, each node in the cluster was statically assigned the capability of running a predefined number of map slots and a predefined number of reduce slots. The slots could not be shared between maps and reduces. This static allocation of slots wasn't optimal because slot requirements vary during the MapReduce application life cycle. Typically, there is a demand for map slots when the job starts, as opposed to the need for reduce slots toward the end of the job.

The resource allocation model in YARN addresses the inefficiencies of static allocations by providing for greater flexibility. As described previously, resources are requested in the form of containers, where each container has a number of nonstatic

attributes. YARN currently has attribute support for memory and CPU. The generalized attribute model can also support things like bandwidth or GPUs. In the future resource management model, only a minimum and a maximum for each attribute are defined, and ApplicationManagers can request containers with attribute values as multiples of the minimum.

## Client Resource Request

A YARN application starts with a client resource request. Figure 4.2 illustrates the initial step in which a client communicates with the ApplicationManager component of the ResourceManager to initiate this process. The client must first notify the ResourceManager that it wants to submit an application. The ResourceManager responds with an ApplicationID and information about the capabilities of the cluster that will aid the client in requesting resources. This process is shown in Steps 1 and 2 in Figure 4.2.

## ApplicationMaster Container Allocation

Next the client responds with a "Application Submission Context" in Step 3. The Application Submission context contains the ApplicationID, user, queue, and other information needed to start the ApplicationMaster. In addition a "Container Launch

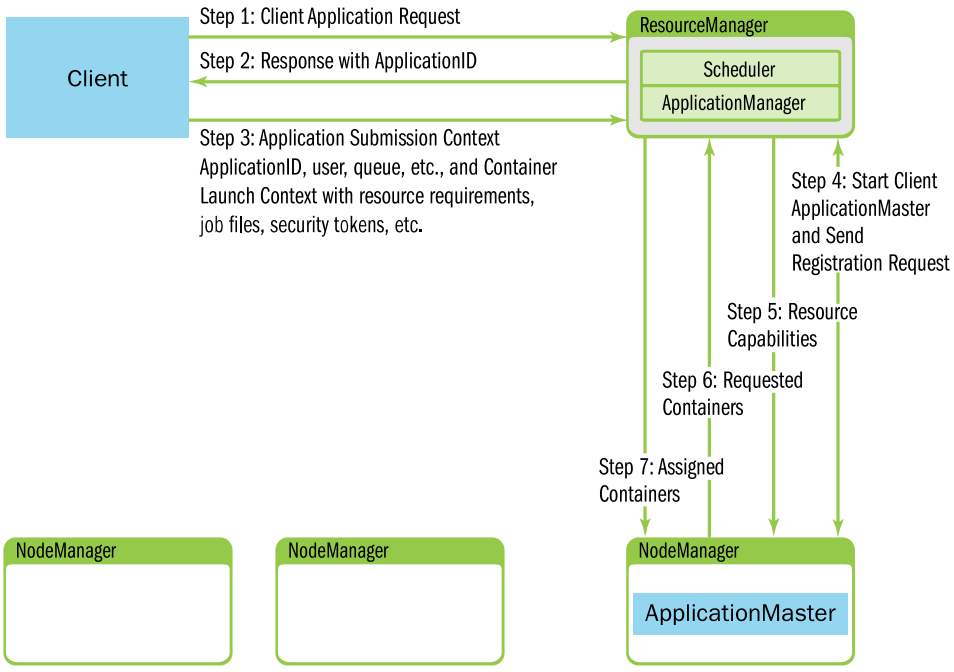



Figure 4.2 Example client resource request to ResourceManager

Context" (CLC) is sent to the ResourceManager. The CLC provides resource requirements (memory/CPU), job files, security tokens, and other information needed to launch an ApplicationMaster on a node. Once the application has been submitted, the client can also request that the ResourceManager kill the application or provide status reports about the application.

When the ResourceManager receives the application submission context from a client, it schedules an available container for the ApplicationMaster. This container is often called "container 0" because it is the ApplicationMaster, which must request additional containers. If there are no applicable containers, the request must wait. If a suitable container can be found, then the ResourceManager contacts the appropriate NodeManager and starts the ApplicationMaster (Step 4 in Figure 4.2). As part of this step, the ApplicationMaster RPC port and tracking URL for monitoring the application's status will be established.

In response to the registration request, the ResourceManager will send information about the minimum and maximum capabilities of the cluster (Step 5 in Figure 4.2). At this point the ApplicationMaster must decide how to use the capabilities that are currently available. Unlike some resource schedulers in which clients request hard limits, YARN allows applications to adapt (if possible) to the current cluster environment.

Based on the available capabilities reported from the ResourceManager, the ApplicationMaster will request a number of containers (Step 6 in Figure 4.2). This request can be very specific, including containers with multiples of the resource minimum values (e.g., extra memory). The ResourceManager will respond, as best as possible based on scheduling policies, to this request with container resources that are assigned to the ApplicationMaster (Step 7 in Figure 4.2).

As a job progresses, heartbeat and progress information is sent from the ApplicationMaster to the ResourceManager (shown in Figure 4.3). Within these heartbeats, it is possible for the ApplicationMaster to request and release containers. When the job finishes, the ApplicationMaster sends a Finish message to the ResourceManager and exits.

## ApplicationMaster–Container Manager Communication

At this point, the ResourceManager has handed off control of assigned NodeManagers to the ApplicationMaster. The ApplicationMaster will independently contact its assigned node managers and provide them with a Container Launch Context that includes environment variables, dependencies located in remote storage, security tokens, and commands needed to start the actual process (refer to Figure 4.3). When the container starts, all data files, executables, and necessary dependencies are copied to local storage on the node. Dependencies can potentially be shared between containers running the application.

Once all containers have started, their status can be checked by the ApplicationMaster. The ResourceManager is absent from the application progress and is free to schedule and monitor other resources. The ResourceManager can direct the NodeManagers to kill containers. Expected kill events can happen when the

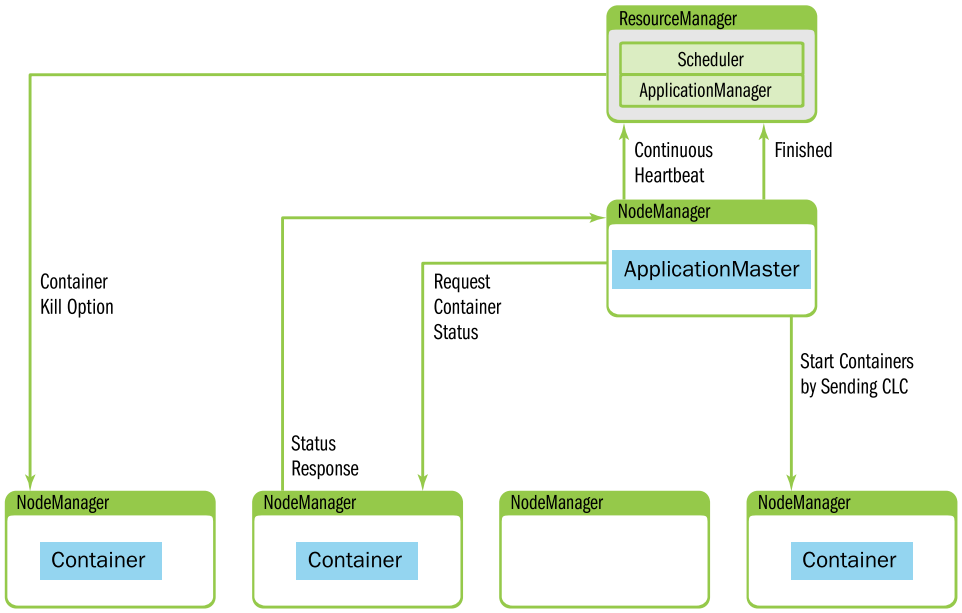



**Figure 4.3** ApplicationMaster NodeManager interaction.

ApplicationMaster informs the ResourceManager of its completion, or the ResourceManager needs nodes for another applications, or the container has exceeded its limits. When a container is killed, the NodeManager cleans up the local working directory. When a job is finished, the ApplicationMaster informs the ResourceManager that the job completed successfully. The ResourceManager then informs the NodeManager to aggregate logs and clean up container-specific files. The NodeManagers are also instructed to kill any remaining containers (including the ApplicationMaster) if they have not already exited.

## Managing Application Dependencies

In YARN, applications perform their work by running containers that map to processes on the underlying operating system. Containers have dependencies on files for execution, and these files are either required at start-up or may be needed one or more times during application execution. For example, to launch a simple Java program as a container, we need a collection of classes and/or a file and potentially more jar files as dependencies. Instead of forcing every application to either access (mostly for reading) these files remotely every time or manage the files themselves, YARN gives the applications the ability to *localize* these files.

When starting a container, an ApplicationMaster can specify all the files that a container will require and, therefore, that should be localized. Once these files

are specified, YARN takes care of the localization and hides all the complications involved in securely copying, managing, and later deleting these files.

In the remainder of this section, we'll explain the basic concepts underlying this functionality.

## LocalResources Definitions

We will begin with some definitions that will aid in the discussion of application dependencies.

- **Localization:** The process of copying or downloading remote resources onto the local file system. Instead of always accessing a resource remotely, that resource is copied to the local machine, so it can then be accessed locally from that point of time.
- **LocalResource:** The file or library required to run a container. The NodeManager is responsible for localizing the resource prior to launching the container. For each LocalResource, applications can specify the following:
  - URL: A remote location from where a LocalResource must be downloaded.
  - Size: The size in bytes of the LocalResource.
  - TimeStamp: The last modification of the resource on the remote file system before container start.
  - LocalResourceType: A specific type of a resource localized by the NodeManager—FILE, ARCHIVE, or PATTERN.
  - Pattern: The pattern that should be used to extract entries from the archive (used only when the type is PATTERN).
  - LocalResourceVisibility: The specific visibility of a resource localized by the NodeManager. The visibility can be either PUBLIC, PRIVATE, or APPLICATION.

  A container can request and use any kind of files for localization, provided they are used as read-only by the containers. Typical examples of LocalResources include the following:
  - Libraries required for starting the container, such as a jar file.
  - Configuration files required to configure the container once started (e.g., remote service URLs, application default configurations).
  - A static dictionary file.

  The following are some examples of bad candidates for LocalResources:

  - Shared files that external components may potentially update in the future and for which current containers wish to track these changes.
  - Files that applications themselves directly want to update.
  - Files through which an application plans to share the updated information with external services.
- **LocalCache:** The NodeManager maintains and manages several local caches of all the files downloaded. The resources are uniquely identified based on the remote URL originally used while copying that file.

## LocalResource Timestamps

As mentioned earlier, NodeManager tracks the last-modification timestamp of each LocalResource before a container starts. Before downloading them, the NodeManager checks that the files haven't changed in the interim. This check ensures a consistent view at the LocalResources—an application can use the very same file contents all the time it runs without worrying about data corruption issues due to concurrent writers to the same file.

Once the file is copied from its remote location to one of the NodeManager's local disks, it loses any connection to the original file other than the URL (used while copying). Any future modifications to the remote file are *not* tracked; hence, if an external system has to update the remote resource, it should be done via versioning. YARN will cause containers that depend on modified remote resources to fail, in an effort to prevent inconsistencies.

Note that the ApplicationMaster specifies the resource timestamps to a NodeManager while starting any container on that node. Similarly, for the container running the ApplicationMaster itself, the client must populate the timestamps for all the resources that ApplicationMaster needs.

In case of a MapReduce application, the MapReduce JobClient determines the modification timestamps of the resources needed by the MapReduce ApplicationMaster. The ApplicationMaster itself then sets the timestamps for the resources needed by the MapReduce tasks.

## LocalResource Types

Each LocalResource can be of one of the following types:

- FILE: A regular file, either textual or binary.
- ARCHIVE: An archive, which is automatically unarchived by the NodeManager. As of now, NodeManager recognizes jar, tar, tar.gz, and .zip files.

- PATTERN: A hybrid of ARCHIVE and FILE types. The original file is retained, and at the same time (only) part of the file is unarchived on the local file system during localization. Both the original file and the extracted files are put in the same directory. Which contents have to be extracted from the ARCHIVE and which shouldn't be are determined by the pattern field in the LocalResource specification. Currently, only jar files are supported under PATTERN type; all others are treated as a regular ARCHIVE.

## LocalResource Visibilities

LocalResources can be of three types depending on their specified LocalResource-Visibility—that is, depending on how visible/accessible they are on the original storage/file system.

### PUBLIC

All the LocalResources (remote URLs) that are marked PUBLIC are accessible for containers of any user. Typically PUBLIC resources are those that can be accessed by anyone on the remote file system and, following the same ACLs, are copied into a public LocalCache. If in the future a container belonging to this or any other application (of this or any user) requests the same LocalResource, it is served from the LocalCache and, therefore, not copied or downloaded again if it hasn't been evicted from the LocalCache by then. All files in the public cache will be owned by "yarn-user" (the user that NodeManager runs as) with world-readable permissions, so that they can be shared by containers from all users whose containers are running on that node.

### PRIVATE

LocalResources that are marked PRIVATE are shared among all applications of the same user on the node. These LocalResources are copied into the specific user's (the user who started the container—that is, the application submitter) private cache. These files are accessible to all the containers belonging to different applications, but all started by the same user. These files on the local file system are owned by the user and are not accessible by any other user. Similar to the public LocalCache, even for the application submitters, there aren't any write permissions; the user cannot modify these files once localized. This feature is intended to avoid accidental write operations to these files by one container that might potentially harm other containers. All containers expect each LocalResource to be in the same state as originally specified (mirroring the original timestamp and/or version number).

### APPLICATION

All the resources that are marked as having the APPLICATION scope are shared only among containers of the same application on the node. They are copied into the application-specific LocalCache that is owned by the user who started the container (application submitter). All of these files are owned by the user with read-only permissions.

#### Specifying LocalResource Visibilities

The ApplicationMaster specifies the visibility of a LocalResource to a NodeManager while starting the container; the NodeManager itself doesn't make any decisions or classify resources. Similarly, for the container running the ApplicationMaster itself, the client has to specify visibilities for all the resources that the ApplicationMaster needs.

In case of a MapReduce application, the MapReduce JobClient decides the resource type which the corresponding ApplicationMaster then forwards to a NodeManager.

### Lifetime of LocalResources

As mentioned previously, different types of LocalResources have different life cycles:

- PUBLIC LocalResources are not deleted once the container or application finishes, but rather are deleted only when there is pressure on each local directory for disk capacity. The threshold for local files is dictated by the configuration property `__yarn.nodemanager.localizer.cache.target-size-mb__`, as described later in this section.
- PRIVATE LocalResources follow the same life cycle as PUBLIC resources.
- APPLICATION-scoped LocalResources are deleted immediately after the application finishes.

For any given application, we may have multiple ApplicationAttempts, and each attempt may start zero or more containers on a given NodeManager. When the first container belonging to an ApplicationAttempt starts, NodeManager localizes files for that application as requested in the container's launch context. If future containers request more such resources, then all of them will be localized. If one ApplicationAttempt finishes or fails and another is started, ResourceLocalizationService doesn't do anything with respect to the previously localized resources. However, when the application finishes, the ResourceManager communicates that information to NodeManagers which in turn clear the application LocalCache. In summary, APPLICATION LocalResources are truly application scoped and not ApplicationAttempt scoped.

## Wrap-up

The three main yarn components work together to deliver a new level of functionality to Apache Hadoop. The ResourceManager acts as a pure scheduler controlling the use of cluster resources in the form of resource containers (e.g., CPUs, memory). User applications are under the control of an application-specific ApplicationMaster (itself a container) that must negotiate the use of additional containers with the ResourceManager at run time. Once the ApplicationMaster has been given resources, it works with the per-node NodeManagers to start and monitor containers on the cluster nodes. Containers are flexible and can be released and requested as the application progresses.

To ensure best utilization of the cluster, administrators have three scheduling options: FIFO, capacity, and fair share. These schedulers are used by the ResourceManager to best match the user needs with the available cluster resources.

LocalResources are new and are a very useful feature that application writers can exploit to declare their start-up and run-time dependencies.

In this new YARN environment, MapReduce does not hold a special place in the workflow because it is "just an application framework" directed by an ApplicationMaster. Other frameworks are available or under development for use in the YARN environment.

# 5

# Installing Apache Hadoop YARN

A cluster-wide installation of Hadoop 2 YARN is necessary to harness the parallel processing capability of the Hadoop ecosystem. HDFS and YARN form the core components of Hadoop version 2. The familiar MapReduce process is still part of YARN, but it has become its own application framework. The installation methods described in this chapter enable you to fully install the base components needed for YARN functionality. Recall that in YARN, the JobTracker has been replaced by the ResourceManager and the per-node TaskTrackers have been replaced by the NodeManager. The basic HDFS installation using a NameNode and DataNodes remains unchanged.

We describe two methods of installation here: a script-based install and a GUI-based install using Apache Ambari. In both cases, a certain minimum amount of user is input required for successful installation.

## The Basics

Of all the many ways to install Hadoop version 2, one of the more involved ways is to simply download the distribution from the Apache Software Foundation (ASF) site. This process requires the creation of a few directories, possibly creation and editing of the configuration files, and maybe even creation of your own scripts. Such a process is a great way to learn the basics of Hadoop administration, but for those wanting a fast and flexible route to Hadoop installation, we present two methods of automated installation in this chapter.

The system requirements for a Hadoop installation are somewhat basic. The installation of the ASF distribution still relies on a Linux file system such as ext3, ext4, or xfs. A Java Development Kit 1.6 (or greater) is required as well. The OpenJDK that comes with most popular Linux distributions should work for most installation procedures. Production systems should have processors, memory, disk, and network

sufficient for the production use cases of each organization. To just get started, however, all we need are Linux servers and the right version of Java.

Hadoop 2 offers significant improvements beyond YARN—namely, improvements in the HDFS (Hadoop File System) that can influence infrastructure decisions. Whether to use NameNode federation and NameNode HA (high availability) are the two important decisions that must be made by most organizations. NameNode federation significantly improves the scalability and performance of HDFS by introducing the ability to deploy multiple NameNodes for a single cluster. In addition, HDFS introduces built-in high availability for the NameNode via a new feature called the Quorum Journal Manager (QJM). QJM-based HA features an active NameNode and a standby NameNode. The standby NameNode can become active either by a manual process or automatically. Automatic failover works in coordination with ZooKeeper. Hadoop 2 HDFS introduces the ZKFailoverController, which uses ZooKeeper's election functionality to determine the active NameNode. Other features are also available in HDFS, but a complete description of HDFS installation and configuration is beyond the scope of this book. More information on HDFS options can be found at http://hadoop.apache.org/docs/stable/hdfs_user_guide.html.

For the purposes of describing the YARN installation process, a simple HDFS installation familiar to Hadoop 1 users will be used. It consists of a single NameNode, a SecondaryNameNode, and multiple DataNodes.

## System Preparation

Once your system requirements are confirmed and you have downloaded the latest version of Hadoop 2, you will need some information that will make the scripted installation easier. The workhorse of this method is the open-source tool Parallel Distributed Shell (http://sourceforge.net/projects/pdsh), commonly referred to as simply `pdsh`, which describes itself as "an efficient, multithreaded remote shell client which executes commands on multiple remote hosts in parallel." In simple terms, `pdsh` will execute commands remotely on hosts specified either on the command line or in a file. As Hadoop is a distributed system that potentially spans thousands of hosts, `pdsh` can be a very valuable tool for managing systems. Also included in the `pdsh` distribution is the `pdcp` command, which performs distributed copying of files. We'll use both the `pdsh` and `pdcp` commands to install Hadoop 2.

> **Note**
>
> The following procedure describes an RPM (Red Hat)-based installation. The scripts described here are available for download from the book repository (see Appendix A), along with instructions for Ubuntu installation.

### Step 1: Install EPEL and pdsh

The `pdsh` tool is easily installed using an existing RPM package. It is also possible to install `pdsh` by downloading prebuilt binaries or by compiling the tool from its source

files. In most cases, this tool will be available through your system's existing software installation or update mechanism. For Red Hat–based systems, this is the yum RPM repository; for SUSE systems, it is the zypper RPM repository.

For the purposes of this installation, we will use Red Hat RPM-based system. A version of the `pdsh` package is distributed in the Extra Packages for Enterprise Linux (EPEL) repository. EPEL has extra packages not in the standard RPM repositories for distributions based on Red Hat Linux. The following steps, performed as root, are needed to install the EPEL repository the pdsh RPM.

```
# rpm -Uvh http://download.fedoraproject.org/pub/epel/6/i386/epel-release-6-
➥8.noarch.rpm
…
# yum install pdsh
```

## Step 2: Generate and Distribute ssh Keys

For `pdsh` to work effectively, we need to configure "password-less" `ssh` (secure shell). When `pdsh` executes remote commands, it will attempt to do so without the need for a password, similar to a user executing an `ssh` command. The first step is to generate the public and private keys for any user executing `pdsh` commands (at a minimum, for the root user). On Linux, the OpenSSH package is generally used for this task. This package includes the `ssh-keygen` command shown later in this subsection. For the easiest installation, as root we execute the `ssh-keygen` command and accept all the defaults, making sure not to enter a passphrase. If we did specify a passphrase, we would need to enter this passphrase every time we used `pdsh` or any other tool that accessed the private key.

After generating the keys, we need to copy the public key to the hosts to which we want to log in via `ssh` without a password. While this might seem like a painstaking task, OpenSSH has the tools to make things easier. The OpenSSH clients package, which is usually installed by default, provides `ssh-copy-id`, a command that copies a public key to another host and adds the host to the `ssh known_hosts` file. During an installation using `pdsh`, we'll want to use the host's fully qualified domain name (FQDN), as this is also the hostname we'll use in Hadoop configuration files.

```
# ssh-keygen -t rsa
…
# ssh-copy-id -i /root/.ssh/id_rsa.pub my.fqdn.tld
…
```

Once `pdsh` is installed and password-less `ssh` is working, the following type of command should be possible. See the `pdsh man` page for more information.

```
# pdsh -w my.fqdn.tld hostname
```

One feature of `pdsh` that will be useful to us is the ability to use host lists. For example, if we create a file called `all_hosts` and include the FQDNs of all the nodes in the cluster, then `pshd` can use this list as follows:

```
# pdsh -w ^all_hosts uptime
```

This feature will be used extensively in the installation script.

# Script-based Installation of Hadoop 2

To simplify the installation process, we will use an installation script to perform the series of tasks required for a typical Hadoop 2 installation. The script is available for review in Appendix B, and can serve as a guide to the steps necessary to install Hadoop 2. The script and all other files are also available from the book repository; see Appendix A for details. The installation script is designed to be flexible and customizable to your needs. Modification for your specific needs is encouraged.

Before we begin, there are some assumptions and some prerequisites that you will need to provide. First, we assume that all nodes have a current Red Hat distribution installed (or Red Hat–like distribution). Second, we assume that adequate memory, cores, network, and disk space are available to meet your needs. Best practices for selecting hardware can be found on the web: http://hortonworks.com/blog/best-practices-for-selecting-apache-hadoop-hardware. Finally, we assume that `pdsh` is working across the cluster.

You will need to choose a version of Hadoop. As of this book's writing, version 2.2.0 was the most current version available from Apache Hadoop's website. To obtain the current version, go to http://hadoop.apache.org/releases.html and follow the links to the current version. For this install, we downloaded `hadoop-2.2.0.tar.gz` (there is no need to use the "src" version).

## JDK Options

There are two options for installing a Java JDK. The first is to install the OpenJDK that is part of the distribution. You can test whether the OpenJDK is installed by issuing the following command. If the OpenJDK is installed, the packages will be listed.

```
# rpm -qa|grep jdk
java-1.6.0-openjdk-devel-1.6.0.0-1.62.1.11.11.90.el6_4.x86_64
java-1.6.0-openjdk-1.6.0.0-1.62.1.11.11.90.el6_4.x86_64
```

Both the base and devel versions should be installed. The other recommended JVM is `jdk-6u31-linux-x64-rpm.bin` from Oracle. This version can be downloaded from http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html.

If you choose to use the Oracle version, make sure you have removed the OpenJDK from all your systems using the following command:

```
# rpm -e java-1.6.0-openjdk-devel java-1.6.0-openjdk-devel
```

You may need to add the "`--nodeps`" option to remove the packages. Keep in mind that if there are dependencies on the OpenJDK, you may need to change some settings to use the Oracle JDK.

## Step 1: Download and Extract the Scripts

The install scripts and supporting files are available from the book repository (see Appendix A). You can simply use `wget` to pull down the `hadoop2-install-scripts.tgz` file.

As root, extract the file move to the `hadoop2-install-scripts` working directory:

```
# tar xvzf hadoop2-install-scripts.tgz
# cd hadoop2-install-scripts
```

If you have not done so already, download your Hadoop version and place it in this directory (do not extract it). Also, if you are using the Oracle SDK, place it in this directory as well.

## Step 2: Set the Script Variables

The main script is called `install-hadoop2.sh`. The following is a list of the user-defined variables that appear at the beginning of this script. You will want to make sure your version matches the `HADOOP_VERSION` variable. You can also change the install path by changing `HADOOP_HOME` (in this case, the path is `/opt`). Next, there are various directories that are used by the various services. The following example assumes that `/var` has sufficient capacity to hold all the Hadoop 2 data and log files. These paths can also be changed to suit your hardware. It is a good idea to keep the default values for `HTTP_STATIC_USER` and `YARN_PROXY_PORT`. Finally, `JAVA_HOME` needs to be defined. If you are using the OpenJDK, make sure this definition corresponds to the OpenJDK path. If you are using the Oracle JDK, then download `jdk-6u31-linux-x64-rpm.bin` to this directly and define `JAVA_HOME` as empty: `JAVA_HOME=""`.

```
# Basic environment variables. Edit as necessary
HADOOP_VERSION=2.2.0
HADOOP_HOME="/opt/hadoop-${HADOOP_VERSION}"
NN_DATA_DIR=/var/data/hadoop/hdfs/nn
SNN_DATA_DIR=/var/data/hadoop/hdfs/snn
DN_DATA_DIR=/var/data/hadoop/hdfs/dn
YARN_LOG_DIR=/var/log/hadoop/yarn
HADOOP_LOG_DIR=/var/log/hadoop/hdfs
HADOOP_MAPRED_LOG_DIR=/var/log/hadoop/mapred
YARN_PID_DIR=/var/run/hadoop/yarn
HADOOP_PID_DIR=/var/run/hadoop/hdfs
HADOOP_MAPRED_PID_DIR=/var/run/hadoop/mapred
HTTP_STATIC_USER=hdfs
YARN_PROXY_PORT=8081
# If using local OpenJDK, it must be installed on all nodes.
# If using jdk-6u31-linux-x64-rpm.bin, then
# set JAVA_HOME="" and place jdk-6u31-linux-x64-rpm.bin in this directory
JAVA_HOME=/usr/lib/jvm/java-1.6.0-openjdk-1.6.0.0.x86_64/
```

### Step 3: Provide Node Names

Once you have set the options, it is time to perform the installation. Keep in mind that the script relies heavily on `pdsh` and `pdcp`. If these two commands are not working across the cluster, then the installation procedure will not work. You can get help for the script by running

```
# ./install-hadoop2.sh -h
```

The script offers two options: file based or interactive. In file-based mode, the script needs the following list of files with the appropriate node names for your cluster:

- `nn_host`: HDFS NameNode hostname
- `rm_host`: YARN ResourceManager hostname
- `snn_host`: HDFS SecondaryNameNode hostname
- `mr_history_host`: MapReduce Job History server hostname
- `yarn_proxy_host`: YARN Web Proxy hostname
- `nm_hosts`: YARN NodeManager hostnames
- `dn_hosts`: HDFS DataNode hostnames, separated by a space

If you use the interactive method, the files will be created automatically. If you choose the file-based method, you can edit the files yourself. With exception of `nm_hosts` and `dn_hosts`, all of the files require one hostname. Depending on your installation, some of these hosts may be the same physical machine. The `nm_hosts` and `dn_hosts` files take a space-delimited list of hostnames, which will identify HDFS data nodes and/or YARN worker nodes.

### Step 4: Run the Script

After you have checked over everything, you can run the script as follows, using `tee` to keep a record of the install:

```
# ./install-hadoop2.sh -f |tee install-hadoop2-results
```

Some steps may take longer than others. If everything worked correctly, a MapReduce job (the classic "calculate pi" example) will be run. If it is successful, the installation process is complete. You may wish to install other tools like Pig, Hive, or HBase as well. For your reference, the script does the following:

1. Copies the Hadoop tar file to all hosts.
2. Optionally copies and installs Oracle JDK 1.6.0_31 to all hosts.
3. Sets the `JAVA_HOME` and `HADOOP_HOME` environment variables on all hosts.
4. Extracts the Hadoop distribution on all hosts.
5. Creates system accounts and groups on all hosts (Group: hadoop, Users: yarn, hdfs, and mapred).
6. Creates HDFS data directories on the NameNode host, SecondaryNameNode host, and DataNode hosts.

7. Creates log directories on all hosts.
8. Creates pid directories on all hosts.
9. Edits Hadoop environment scripts for log directories on all hosts.
10. Edits Hadoop environment scripts for pid directories on all hosts.
11. Creates the base Hadoop XML config files (`core-site.XML`, `hdfs-site.XML`, `mapred-site.XML`, `yarn-site.XML`).
12. Copies the base Hadoop XML config files to all hosts.
13. Creates configuration, command, and script links on all hosts.
14. Formats the NameNode.
15. Copies start-up scripts to all hosts (hadoop-datanode, hadoop-historyserver, hadoop-namenode, hadoop-nodemanager, hadoop-proxyserver, hadoop-resourcemanager, and hadoop-secondarynamenode).
16. Starts Hadoop services on all hosts.
17. Creates MapReduce Job History directories.
18. Runs the YARN pi MapReduce job.

## Step 5: Verify the Installation

There are a few points in the script-based installation process where problems may occur. One important step is formatting the NameNode (Step 14 in the preceding list). The results of this command will show up in the script output. Make sure there were no errors with this command.

If you note other errors, such as when starting the Hadoop daemons, check the log files located under `$HADOOP_HOME/logs`. The final part of the script runs the following example "pi" MapReduce job command.

```
hadoop jar \
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-$HADOOP_VERSION.jar \
pi -Dmapreduce.clientfactory.class.name=org.apache.hadoop.mapred.YarnClientFactory \
-libjars $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-jobclient-
➥$HADOOP_VERSION.jar \
16 10000
```

If this test is successful, you should see the following at the end of the output. (Your run-time time will vary.)

```
Job Finished in 25.854 seconds
Estimated value of Pi is 3.14127500000000000000
```

You can also examine the HDFS file system using the following command:

```
# hdfs dfs -ls /
Found 6 items
drwxr-xr-x   - hdfs   hdfs            0 2013-02-06 21:17 /apps
drwxr-xr-x   - hdfs   hadoop          0 2013-02-06 22:26 /benchmarks
```

```
drwx------   - mapred hdfs            0 2013-04-25 16:20 /mapred
drwxr-xr-x   - hdfs   hdfs            0 2013-05-16 11:53 /system
drwxrwxr--   - hdfs   hadoop          0 2013-05-14 14:54 /tmp
drwxrwxr-x   - hdfs   hadoop          0 2013-04-26 02:00 /user
```

Similar to Hadoop version 1, many aspects of Hadoop version 2 are available via the built-in web UI. The web UI can be accessed by directly accessing the following URL in your favorite browser or by issuing the following command as the user hdfs (using your local *hostname*):

```
$ firefox http://hostname:8088/cluster
```

An example of the UI is shown in Figure 5.1.

The test job should be listed in the UI window. You can find out about the job history by clicking the History link on the right side of the job summary. When you do so, the window shown in Figure 5.2 should be displayed.

To check the status of the your parallel file system, enter *hostname*:50070 into the browser. A page similar to Figure 5.3 should be displayed.

> **Note**
>
> The output with Hadoop version 2.2.0 may look slightly different than it does in Figures 5.1, 5.2, and 5.3.

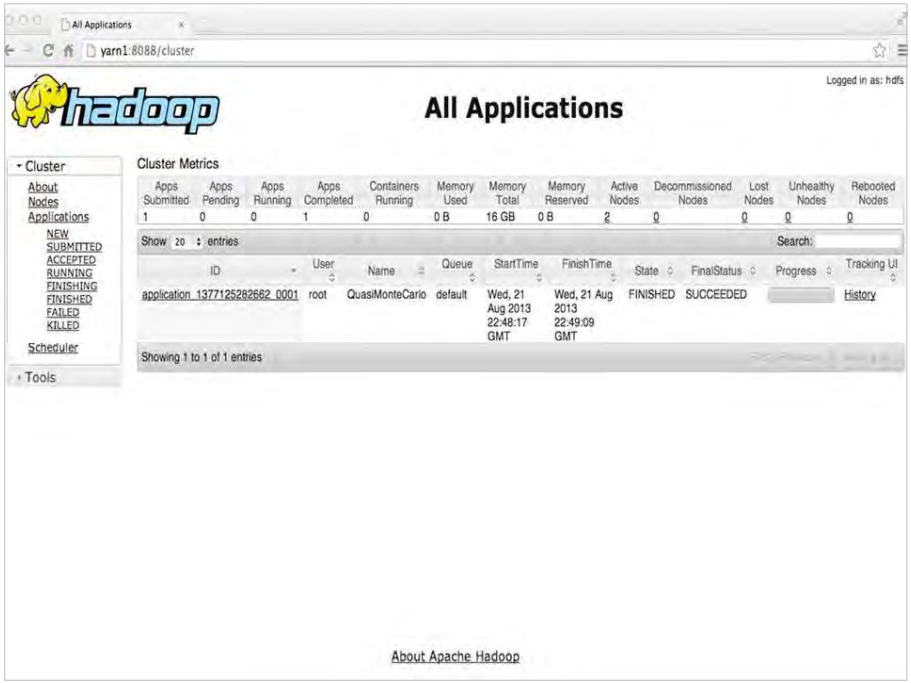



**Figure 5.1** YARN web UI

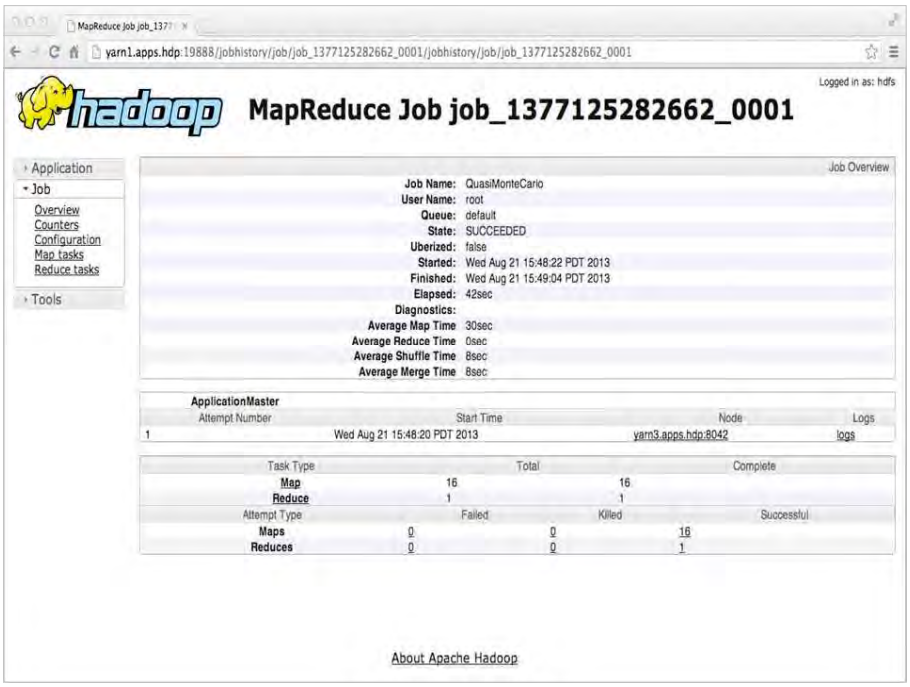



Figure 5.2 YARN pi example job history




Figure 5.3 HDFS NameNode web UI

If the HDFS file system looks fine and there are no dead nodes, then your Hadoop cluster should be fully functional. You can investigate other aspects of your Hadoop 2 cluster by exploring some of the links on both the HDFS and YARN UI pages.

# Script-based Uninstall

To uninstall the Hadoop2 installation, use the `uninstall-hadoop2.sh` script. Make sure any changes you made to the `install-hadoop2.sh` script, such as the Hadoop version number (`HADOOP_VERSION`), are copied to the uninstall script.

# Configuration File Processing

The install script provides some commands that you will find useful for processing Hadoop XML files. If you examine the `install-hadoop2.sh` script, you should notice that several commands are used to create the Hadoop XML configuration files:

```
create_config() --file filename
put_config --file filename --property-property pname --value pvalue
del_config()--file filename --property-property pname
```

where *`filename`* is the name of the XML file, *`pname`* is the property to be defined, and *`pvalue`* is the actual value. The functions are defined in `hadoop-xml-conf.sh`, which is part of the `hadoop2-install-scripts.tgz` archive. See the book repository (Appendix A) for download instructions. Basically, these scripts facilitate the creation of Hadoop XML configuration files. If you want to customize the installation or make changes to the XML files, these scripts may be useful.

# Configuration File Settings

The following is a brief description of the settings specified in the Hadoop configuration files by the installation script. These files are located in `$HADOOP_HOME/etc/hadoop`.

## core-site.xml

In this file, we define two essential properties for the entire system.

```
hdfs://$nn:9000 --> fs.default.name
$HTTP_STATIC_USER --> hadoop.http.staticuser.user
```

First, we define the name of the default file system we wish to use. Because we are using HDFS, we will set this value to `hdfs://$nn:9000` (`$nn` is the NameNode

we specified in the script and `9000` is the standard HDFS port). Next we add the `hadoop.http.staticuser.user` (hdfs) that we defined in the install script. This login is used as the default user for the built-in web user interfaces.

## hdfs-site.xml

The `hdfs-site.xml` file holds information about the Hadoop HDFS file system. Most of these values were set at the beginning of the script. They are copied as follows:

```
$NN_DATA_DIR --> dfs.namenode.name.dir
$SNN_DATA_DIR --> fs.checkpoint.dir
$SNN_DATA_DIR --> fs.checkpoint.edits.dir
$DN_DATA_DIR --> dfs.datanode.data.dir
```

The remaining two values are set to the standard default port numbers (`$nn` is the NameNode and `$snn` is the SecondaryNameNode we input to the script):

```
$nn:50070 --> dfs.namenode.http-address
$snn:50090 --> dfs.namenode.secondary.http-address
```

## mapred-site.xml

Users who are familiar with Hadoop version 1 may notice that this is a known configuration file. Given that MapReduce is just another YARN framework, it needs its own configuration file. The script specifies the following settings:

```
yarn --> mapreduce.framework.name
$mr_hist:10020 --> mapreduce.jobhistory.address
$mr_hist:19888 --> mapreduce.jobhistory.webapp.address
/mapred --> yarn.app.mapreduce.am.staging-dir
```

The first property is `mapreduce.framework.name`. For this property, there are three valid values: local (default), classic, or yarn. Specifying "local" for this value means that the MapReduce Application is run locally in a process on the client machine, without using any cluster resources. This local process will execute the map and reduce tasks for a given job; because it is local, it doesn't need to shuffle data from map task output on one server to reduce task input on another server. Typically, this means that there will be one map task and one reduce task.

Specifying "classic" for the `mapreduce.framework.name` property is appropriate when there is a Hadoop 1.x JobTracker running in your cluster where Hadoop can submit the job. This property exists to accommodate unforeseen situations where there are backward-compatibility problems with users' MapReduce jobs and the need for a classic job submission process to a JobTracker is unavoidable.

As we are interested in using yarn, we will set `mapreduce.framework.name` to "yarn" and use the new MapReduce framework provided by YARN.