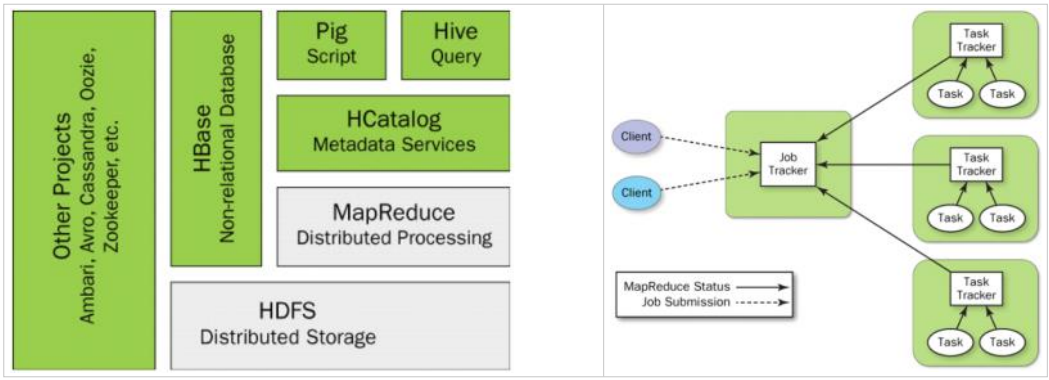
Apache Hadoop YARN Overview

Friday, August 15, 2014 13:34

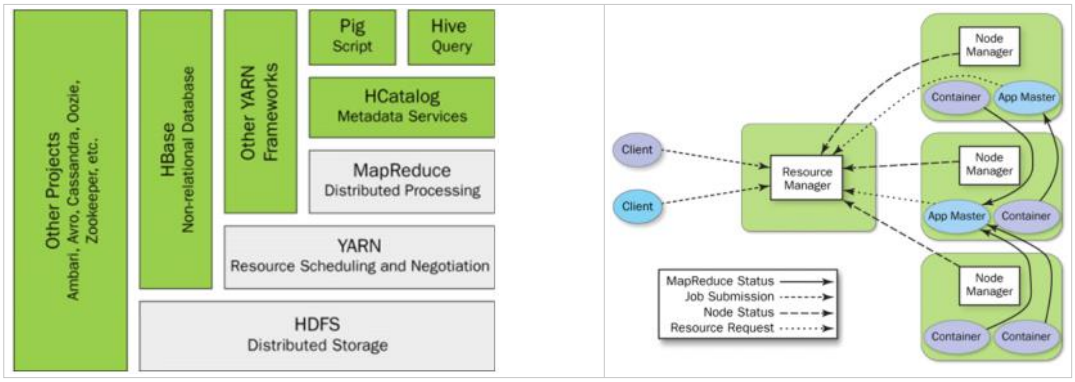
The acronym YARN is short for "Yet Another Resource Negotiator," which is a good description of what YARN actually does. Fundamentally, YARN is a resource scheduler designed to work on existing and new Hadoop clusters.

Beyond MapReduce

The basic structure of Hadoop with Apache Hadoop MapReduce version 1 (MRv1) can be seen in below left Figure. 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. 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.



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. (Refer to above below figures). **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).**



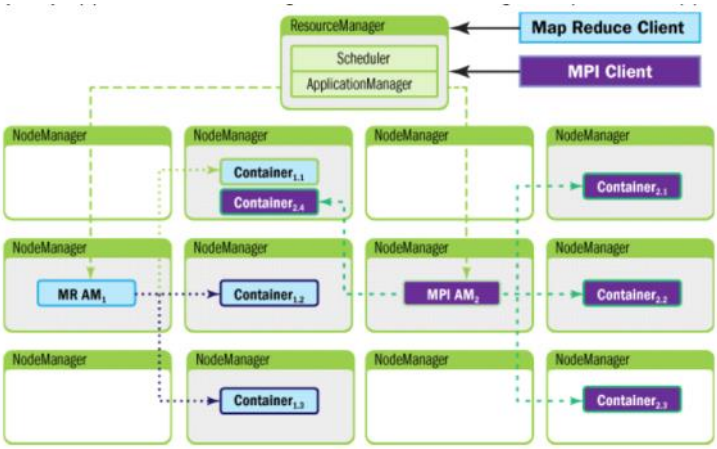
Architecture Overview

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.

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.

Below figure 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 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.



YARN Components

1. **Resource Manager**
   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 YARN, ResourceRequests can be strict or negotiable. This feature provides ApplicationMasters with a great deal of flexibility on how to fulfill the reclamation requests. 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.

2. **YARN Scheduling Components**
   - **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."
   - **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!
   - **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.

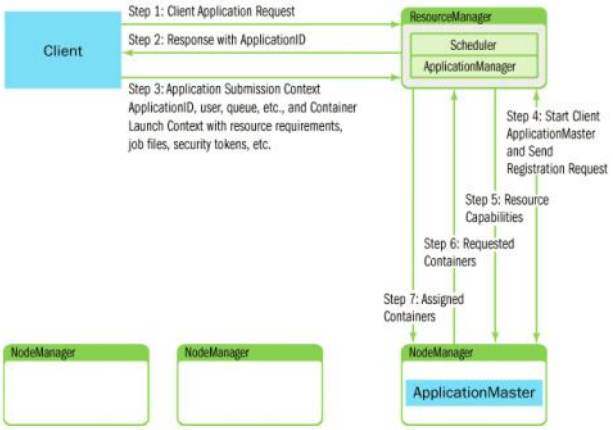
3. **Containers**
   At the fundamental level, a container is a collection of physical resources such as RAM, CPU cores, and disks on a single node. There can be multiple containers on a single node (or a single large one). Every node in the system is considered to be composed of multiple containers 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.

4. **Node Manager**
   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.

5. **Application Master (AM)**
   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, Map-Reduce is just one application framework; this design permits building and deploying distributed applications using other frameworks. 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.

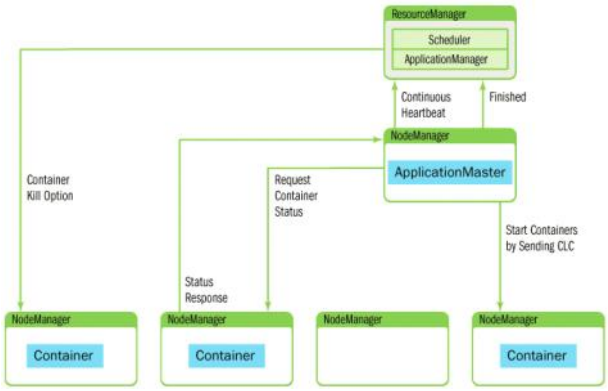
YARN Resource Model

The resource allocation model in YARN addresses the inefficiencies of static allocations by providing for greater flexibility. YARN currently has attribute support for memory and CPU. The generalized attribute model can also support things like bandwidth or GPUs.



1. 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.
2. The ResourceManager responds with an ApplicationID and information about the capabilities of the cluster that will aid the client in requesting resources.
3. 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 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.
4. 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. As part of this step, the ApplicationMaster RPC port and tracking URL for monitoring the application's status will be established.
5. In response to the registration request, the ResourceManager will send information about the minimum and maximum capabilities of the cluster. 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.
6. Based on the available capabilities reported from the ResourceManager, the ApplicationMaster will request a number of containers. This request can be very specific, including containers with multiples of the resource minimum values (e.g., extra memory).
7. The ResourceManager will respond, as best as possible based on scheduling policies, to this request with container resources that are assigned to the ApplicationMaster

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 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. 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.

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. 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 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). PRIVATE LocalResources follow the same life cycle as PUBLIC resources.
- 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. APPLICATION-scoped LocalResources are deleted immediately after the application finishes.