Source

http://hortonworks.com/blog/apache-hadoop-yarn-background-and-an-overview/

http://hortonworks.com/blog/apache-hadoop-yarn-concepts-and-applications/

http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/

http://hortonworks.com/blog/apache-hadoop-yarn-nodemanager/

Apache Hadoop YARN – Background & Overview

Celebrating the significant milestone that was Apache Hadoop YARN being promoted to a full-fledged sub-project of Apache Hadoop in the ASF we present the ***first blog in a multi-part series*** *on Apache Hadoop YARN – a general-purpose, distributed, application management framework that supersedes the classic Apache Hadoop MapReduce framework for processing data in Hadoop clusters.*

MapReduce – The Paradigm

Essentially, the MapReduce model consists of a first, embarrassingly parallel, *map phase* where input data is split into discreet chunks to be processed. It is followed by the second and final *reduce phase* where the output of the map phase is aggregated to produce the desired result. The simple, and fairly restricted, nature of the programming model lends itself to very efficient and extremely large-scale implementations across thousands of cheap, commodity nodes.

Apache Hadoop MapReduce is the most popular open-source implementation of the MapReduce model.

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

One of the keys to this is the ***lack of data motion*** i.e. move compute to data and do not move data to the compute node via the network. Specifically, the MapReduce tasks can be scheduled on the same physical nodes on which data is resident in HDFS, which exposes the underlying storage layout across the cluster. This significantly reduces the network I/O patterns and keeps most of the I/O on the local disk or within the same rack – a core advantage.

Apache Hadoop MapReduce, circa 2011 – A Recap

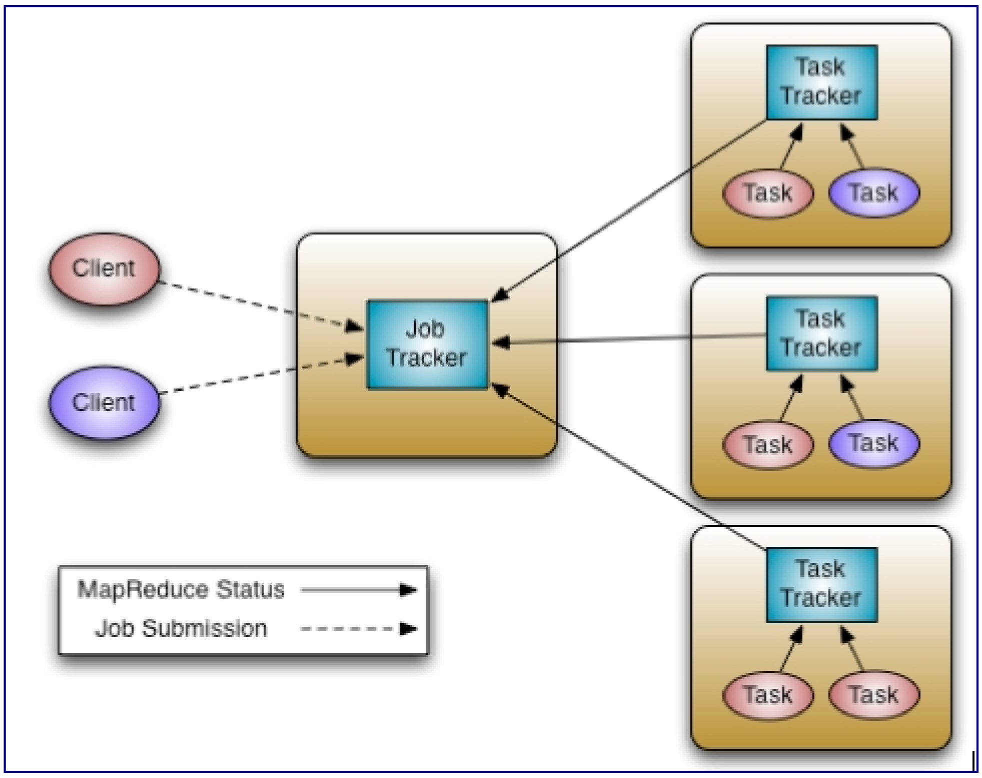
Apache Hadoop MapReduce is an open-source, Apache Software Foundation project, which is an implementation of the MapReduce programming paradigm described above. Now, as someone who has spent over six years working full-time on Apache Hadoop, I normally like to point out that the Apache Hadoop MapReduce project itself can be broken down into the following major facets:

- The end-user ***MapReduce API*** for programming the desired MapReduce application.

- The ***MapReduce framework,*** which is the runtime implementation of various phases such as the map phase, the sort/shuffle/merge aggregation and the reduce phase.
- The ***MapReduce system***, which is the backend infrastructure required to run the user's MapReduce application, manage cluster resources, schedule thousands of concurrent jobs etc.

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

The current Apache Hadoop MapReduce System is composed of the JobTracker, which is the master, and the per-node slaves called TaskTrackers.



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

The TaskTracker has simple responsibilities – launch/teardown tasks on orders from the JobTracker and provide task-status information to the JobTracker periodically.

For a while, we have understood that the Apache Hadoop MapReduce framework needed an overhaul. In particular, with regards to the JobTracker, we needed to address several aspects regarding scalability, cluster utilization, ability for customers to control upgrades to the stack i.e. *customer agility* and equally importantly, supporting workloads other than MapReduce itself.

We've done running repairs over time, including recent support for JobTracker availability and resiliency to HDFS issues (both of which are available in Hortonworks Data Platform v1 i.e. HDP1) but lately they've come at an ever-increasing maintenance cost and yet, did not address core issues such as support for non-MapReduce and customer agility.

Why support non-MapReduce workloads?

MapReduce is great for many applications, but not everything; other *programming models* better serve requirements such as graph processing (Google Pregel / Apache Giraph) and iterative modeling (MPI). When all the data in the enterprise is *already available* in Hadoop HDFS having multiple paths for processing is critical.

Furthermore, since MapReduce is essentially batch-oriented, support for real-time and near real-time processing such as stream processing and CEPFresil are emerging requirements from our customer base.

Providing these within Hadoop enables organizations to see an increased return on the Hadoop investments by lowering operational costs for administrators, reducing the need to move data between Hadoop HDFS and other storage systems etc.

Why improve scalability?

Moore's Law… Essentially, at the same price-point, the processing power available in data-centers continues to increase rapidly. As an example, consider the following definitions of commodity servers:

- 2009 – 8 cores, 16GB of RAM, 4x1TB disk
- 2012 – 16+ cores, 48-96GB of RAM, 12x2TB or 12x3TB of disk.

Generally, at the same price-point, servers are twice as capable today as they were 2-3 years ago – on every single dimension. Apache Hadoop MapReduce is known to scale to production deployments of ~5000 nodes of hardware of 2009 vintage. Thus, ongoing scalability needs are ever present given the above hardware trends.

What are the common scenarios for low cluster utilization?

In the current system, JobTracker views the cluster as composed of nodes (managed by individual TaskTrackers) with **distinct map slots and reduce slots**, which are not *fungible*. Utilization issues occur because maps slots might be 'full' while reduce slots are empty (and vice-versa). Fixing this was necessary to ensure the entire system could be used to its maximum capacity for high utilization.

What is the notion of customer agility?

In real-world deployments, Hadoop is very commonly deployed as a shared, multi-tenant system. As a result, changes to the Hadoop software stack affect a large cross-section if not the entire enterprise. Against that backdrop, customers are very keen on controlling upgrades to the software stack as it has a direct impact on their applications. Thus, allowing multiple, if limited, versions of the **MapReduce framework** is critical for Hadoop.

Enter Apache Hadoop YARN

The fundamental idea of YARN is to split up the two major responsibilities of the JobTracker i.e. resource management and job scheduling/monitoring, into separate daemons: a global

ResourceManager and per-application ApplicationMaster (AM).

The ResourceManager and per-node slave, the NodeManager (NM), form the new, and generic, **system** for managing applications in a distributed manner.

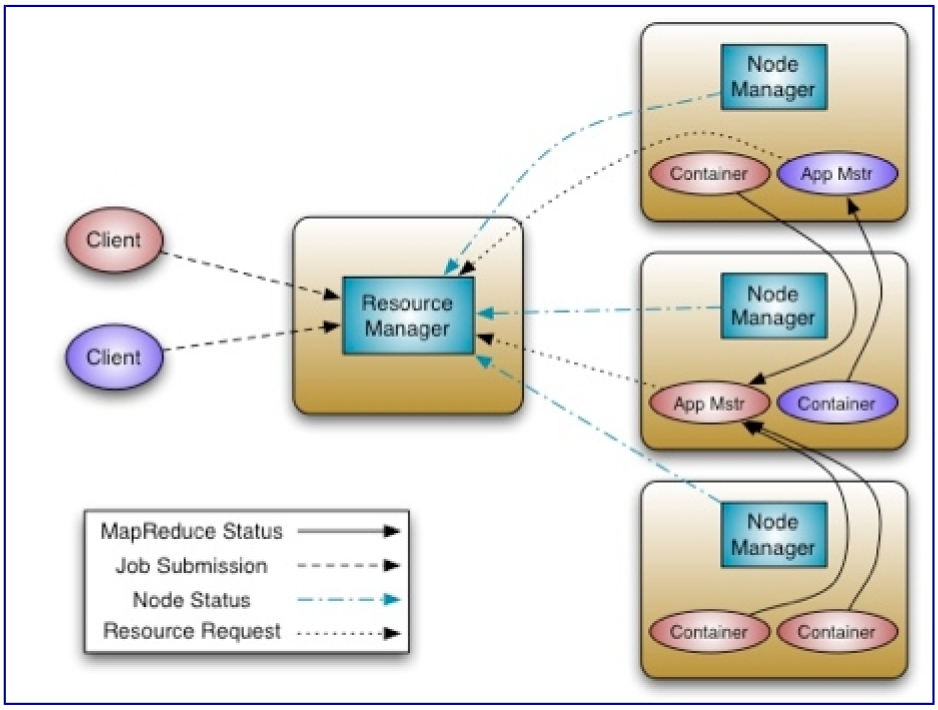
The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application ApplicationMaster is, in effect, a *framework specific* entity and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to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

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

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

The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress. From the system perspective, the ApplicationMaster itself runs as a normal *container*.

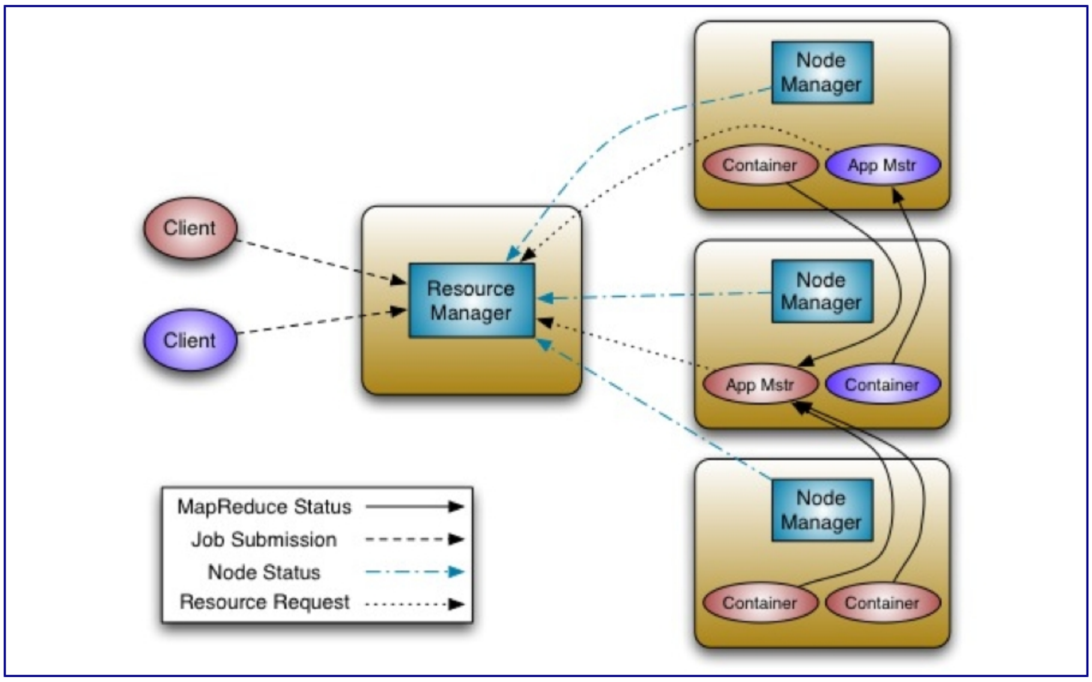
Here is an architectural view of YARN:



One of the crucial implementation details for MapReduce within the new YARN **system** that I'd like to point out is that we have reused the existing MapReduce **framework** without any major surgery. This was very important to ensure **compatibility** for existing MapReduce applications and users. More on this later.

Apache Hadoop YARN – Concepts & Applications

As previously described, YARN is essentially a system for managing distributed applications. It consists of a central **ResourceManager**, which arbitrates all available cluster resources, and a per-node **NodeManager**, which takes direction from the ResourceManager and is responsible for managing resources available on a single node.



Resource Manager

In YARN, the ResourceManager is, primarily, a *pure scheduler*. In essence, it's strictly limited to arbitrating available resources in the system among the competing applications – a *market maker* if you will. It optimizes for cluster utilization (keep all resources in use all the time) against various constraints such as capacity guarantees, fairness, and SLAs. To allow for different policy constraints the ResourceManager has a *pluggable scheduler* that allows for different algorithms such as capacity and fair scheduling to be used as necessary.

ApplicationMaster

Many will draw parallels between YARN and the existing Hadoop MapReduce system (MR1 in Apache Hadoop 1.x). However, the key difference is the *new concept* of an **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 appropriate resource containers from the ResourceManager, tracking their status and monitoring progress.

The ApplicationMaster allows YARN to exhibit the following key characteristics:

- Scale: The Application Master provides much of the functionality of the traditional ResourceManager so that the entire system can scale more dramatically. In tests, we've

already successfully simulated 10,000 node clusters composed of modern hardware without significant issue. This is one of the key reasons that we have chosen to design the ResourceManager as a *pure scheduler* i.e. it doesn't attempt to provide fault-tolerance for resources. We shifted that to become a primary responsibility of the ApplicationMaster instance. Furthermore, since there is an instance of an ApplicationMaster per application, the ApplicationMaster itself isn't a common bottleneck in the cluster.
- Open: Moving all application framework specific code into the ApplicationMaster generalizes the system so that we can now support multiple frameworks such as MapReduce, MPI and Graph Processing.

It's a good point to interject some of the 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.*
- *Since it is essentially user-code, do not trust the ApplicationMaster(s) i.e. any ApplicationMaster is not 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.*

It's useful to remember that, 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. launch HBase in YARN via an hypothetical 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:

- Resource-name (hostname, rackname – we are in the process of generalizing this further to support more complex network topologies with YARN-18).
- Memory (in MB)
- CPU (cores, for now)
- In future, expect us to add more resource-types such as disk/network I/O, GPUs etc.

ResourceRequest and Container

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

In order to meet those goals, the central Scheduler (in the ResourceManager) has extensive information about an application's resource needs, which allows it to make better scheduling decisions across all applications in the cluster. This leads 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 granting a *container,* which satisfies the requirements laid out by the ApplicationMaster in the initial ResourceRequest.

Let's look at the ResourceRequest – it has the following form:

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

Let's walk through each component of the ResourceRequest to understand this better.

- resource-name is either hostname, rackname or * to indicate no preference. In future, we expect to support even more complex topologies for virtual machines on a host, more complex networks etc.
- priority is intra-application priority for this request (to stress, this isn't across multiple applications).
- resource-requirement is required capabilities such as memory, cpu etc. (at the time of writing YARN only supports memory and cpu).
- number-of-containers is just a multiple of such *containers*.

Now, on to the Container.

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* (memory, cpu etc.) on a *specific host*.

The ApplicationMaster has to 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. Of course, the Container allocation is verified, in the secure mode, to ensure that ApplicationMaster(s) *cannot fake allocations* in the cluster.

Container Specification during Container Launch

While a Container, as described above, is merely a *right* to use a specified amount of resources on a specific machine (NodeManager) in the cluster, the ApplicationMaster has to provide considerably more information to the NodeManager to actually *launch* the container.

YARN allows applications to launch any process and, unlike existing Hadoop MapReduce in hadoop-1.x (aka MR1), it isn't limited to Java applications alone.

The YARN Container launch specification API is platform agnostic and contains:

- 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, auxiliary data files etc.
- Security-related tokens.

This 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 (e.g. KVMs).

YARN – Walkthrough

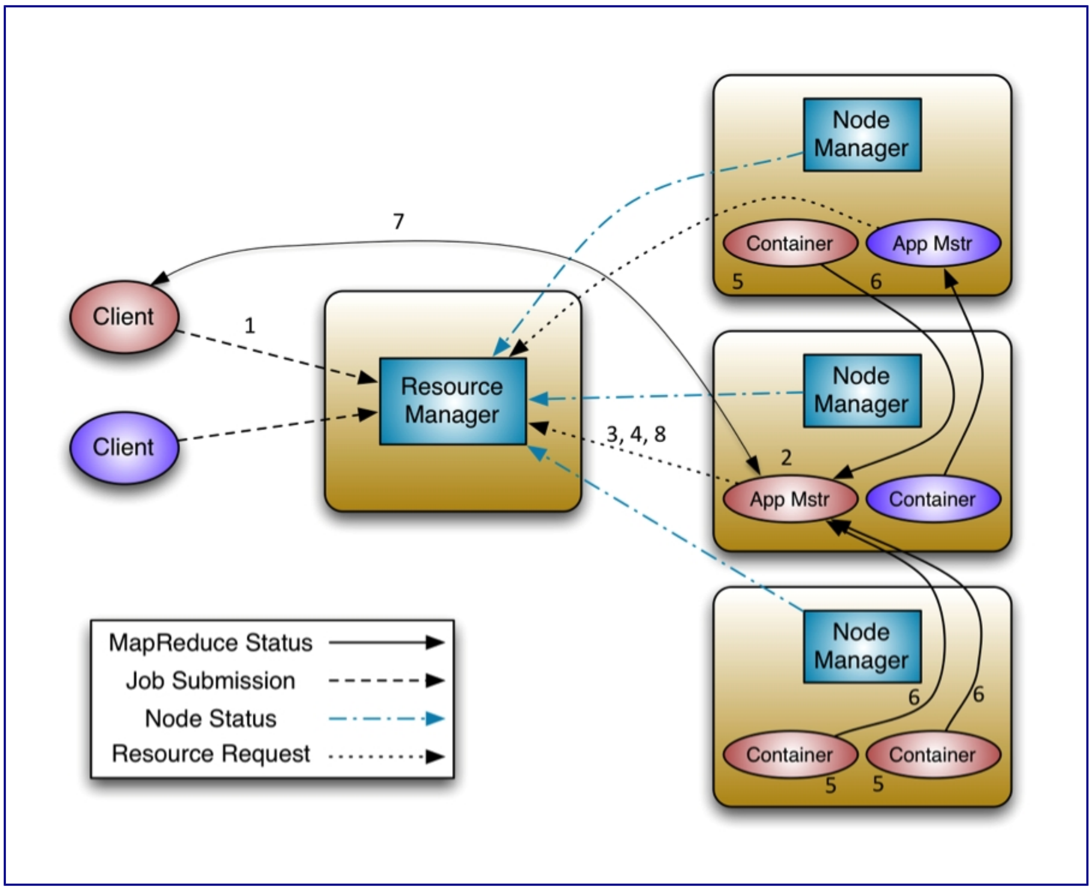
Armed with the knowledge of the above concepts, it will be useful to sketch how applications conceptually work in YARN.

Application execution consists of the following steps:

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

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

1. A client program *submits* the application, including the necessary specifications to *launch the application-specific ApplicationMaster* itself.
2. The ResourceManager assumes the responsibility to negotiate a specified container in which to start the ApplicationMaster and then *launches* the ApplicationMaster.
3. The ApplicationMaster, on boot-up, *registers* with the ResourceManager – the registration allows the client program to query the ResourceManager for details, which allow it to directly communicate with its own ApplicationMaster.
4. During normal operation the ApplicationMaster negotiates appropriate resource containers via the resource-request protocol.
5. On successful container allocations, the ApplicationMaster launches the container by providing the container launch specification to the NodeManager. The launch specification, typically, includes the necessary information to allow the container to communicate with the ApplicationMaster itself.
6. The application code executing within the container then provides necessary information (progress, status etc.) to its ApplicationMaster via an *application-specific protocol*.
7. During the application execution, the client that submitted the program communicates directly with the ApplicationMaster to get status, progress updates etc. via an application-specific protocol.
8. Once the application is complete, and all necessary work has been finished, the ApplicationMaster deregisters with the ResourceManager and shuts down, allowing its own container to be repurposed.

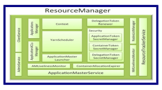


In our next post in this series we dive more into guts of the YARN system, particularly the ResourceManager – stay tuned!

Apache Hadoop YARN – ResourceManager

As previously described, **ResourceManager (RM)** is the master that arbitrates all the available cluster resources and thus helps manage the distributed applications running on the YARN system. It works together with the per-node **NodeManagers (NMs)** and the per-application **ApplicationMasters (AMs)**.

1. **NodeManagers** take instructions from the ResourceManager and manage resources available on a single node.
2. **ApplicationMasters** are responsible for negotiating resources with the ResourceManager and for working with the NodeManagers to start the containers.



ResourceManager Components

The ResourceManager has the following components (see the figure above):

1. **Components interfacing RM to the clients:**

   - **ClientService**: The client interface to the Resource Manager. This component handles all the RPC interfaces to the RM from the clients including operations like application submission, application termination, obtaining queue information, cluster statistics etc.

- **AdminService**: To make sure that admin requests don't get starved due to the normal users' requests and to give the operators' commands the higher priority, all the admin operations like refreshing node-list, the queues' configuration etc. are served via this separate interface.

2. **Components connecting RM to the nodes:**

   - **ResourceTrackerService**: This is the component that responds to RPCs from all the nodes. It is responsible for registration of new nodes, rejecting requests from any invalid/decommissioned nodes, obtain node-heartbeats and forward them over to the YarnScheduler. It works closely with NMLivelinessMonitor and NodesListManager described below.

   - **NMLivelinessMonitor**: To keep track of live nodes and specifically note down the dead nodes, this component keeps track of each node's its last heartbeat time. Any node that doesn't heartbeat within a configured interval of time, by default 10 minutes, is deemed dead and is expired by the RM. All the containers currently running on an expired node are marked as dead and no new containers are scheduling on such node.

   - **NodesListManager**: A collection of valid and excluded nodes. Responsible for reading the host configuration files specified via `yarn.resourcemanager.nodes.include-path` and `yarn.resourcemanager.nodes.exclude-path` and seeding the initial list of nodes based on those files. Also keeps track of nodes that are decommissioned as time progresses.

3. **Components interacting with the per-application AMs:**

   - **ApplicationMasterService**: This is the component that responds to RPCs from all the AMs. It is responsible for registration of new AMs, termination/unregister-requests from any finishing AMs, obtaining container-allocation & deallocation requests from all running AMs and forward them over to the YarnScheduler. This works closely with AMLivelinessMonitor described below.

   - **AMLivelinessMonitor**: To help manage the list of live AMs and dead/non-responding AMs, this component keeps track of each AM and its last heartbeat time. Any AM that doesn't heartbeat within a configured interval of time, by default 10 minutes, is deemed dead and is expired by the RM. All the containers currently running/allocated to an AM that gets expired are marked as dead. RM schedules the same AM to run on a new container, allowing up to a maximum of 4 such attempts by default.

4. **The core of the ResourceManager – the scheduler and related components:**

- **ApplicationsManager**: Responsible for maintaining a collection of submitted applications. Also keeps a cache of completed applications so as to serve users' requests via web UI or command line long after the applications in question finished.

- **ApplicationACLsManager**: RM needs to gate the user facing APIs like the client and admin requests to be accessible only to authorized users. This component maintains the ACLs lists per application and enforces them whenever an request like killing an application, viewing an application status is received.

- **ApplicationMasterLauncher**: Maintains a thread-pool to launch AMs of newly submitted applications as well as applications whose previous AM attempts exited due to some reason. Also responsible for cleaning up the AM when an application has finished normally or forcefully terminated.

- **YarnScheduler**: The Scheduler is responsible for allocating resources to the various running applications subject to constraints of capacities, queues etc. It performs its scheduling function based on the resource requirements of the applications such as memory, CPU, disk, network etc. Currently, only memory is supported and support for CPU is close to completion.

- **ContainerAllocationExpirer**: This component is in charge of ensuring that all allocated containers are used by AMs and subsequently launched on the correspond NMs. AMs run as untrusted user code and can potentially hold on to allocations without using them, and as such can cause cluster under-utilization. To address this, ContainerAllocationExpirer maintains the list of allocated containers that are still not used on the corresponding NMs. For any container, if the corresponding NM doesn't report to the RM that the container has started running within a configured interval of time, by default 10 minutes, the container is deemed as dead and is expired by the RM.

5. **TokenSecretManagers (for security):**ResourceManager has a collection of SecretManagers which are charged with managing tokens, secret-keys that are used to authenticate/authorize requests on various RPC interfaces. A future post on YARN security will cover a more detailed descriptions of the tokens, secret-keys and the secret-managers but a brief summary follows:

    - **ApplicationTokenSecretManager**: To avoid arbitrary processes from sending RM scheduling requests, RM uses the per-application tokens called ApplicationTokens. This component saves each token locally in memory till application finishes and uses it to authenticate any request coming from a valid AM process.

    - **ContainerTokenSecretManager**: SecretManager for ContainerTokens that are special tokens issued by RM to an AM for a container on a specific node. ContainerTokens are used by AMs to create a connection to the corresponding NM where the container is allocated. This component is RM-specific, keeps track of the underlying master and secret-keys and rolls the keys every so often.

   - **RMDelegationTokenSecretManager**: A ResourceManager specific delegation-token secret-manager. It is responsible for generating delegation tokens to clients which can be passed on to unauthenticated processes that wish to be able to talk to RM.

6. **DelegationTokenRenewer:** In secure mode, RM is Kerberos authenticated and so provides the service of renewing file-system tokens on behalf of the applications. This component renews tokens of submitted applications as long as the application runs and till the tokens can no longer be renewed.

Conclusion

In YARN, the ResourceManager is primarily limited to scheduling i.e. only arbitrating available resources in the system among the competing applications and not concerning itself with per-application state management. Because of this clear separation of responsibilities coupled with the modularity described above, and with the powerful scheduler API discussed in the previous post, RM is able to address the most important design requirements – scalability, support for alternate programming paradigms.

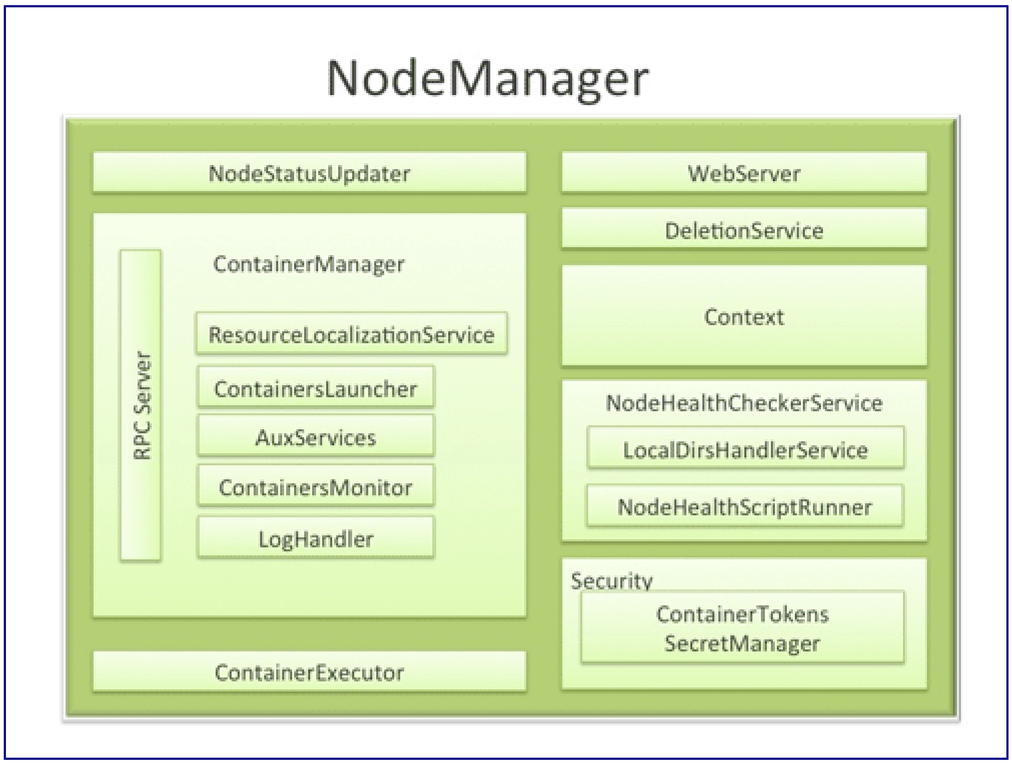
To allow for different policy constraints, the scheduler described above in the RM is pluggable and allows for different algorithms. In a future post of this series, we will dig deeper into various features of CapacityScheduler that schedules containers based on capacity guarantees and queues.

The next post will dive into details of the NodeManager, the component responsible for managing the containers' life cycle and much more.

Apache Hadoop YARN – NodeManager

The NodeManager (NM) is YARN's per-node agent, and takes care of the individual compute nodes in a Hadoop cluster. This includes keeping up-to date with the ResourceManager (RM), overseeing containers' life-cycle management; monitoring resource usage (memory, CPU) of individual containers, tracking node-health, log's management and auxiliary services which may be exploited by different YARN applications.

NodeManager Components



1. **NodeStatusUpdater**

On startup, this component registers with the RM and sends information about the resources available on the nodes. Subsequent NM-RM communication is to provide updates on container statuses – new containers running on the node, completed containers, etc.

In addition the RM may signal the NodeStatusUpdater to potentially kill already running containers.

1. **ContainerManager**

This is the core of the NodeManager. It is composed of the following sub-components, each of which performs a subset of the functionality that is needed to manage containers running on the node.

a. **RPC server**: ContainerManager accepts requests from Application Masters (AMs) to start new containers, or to stop running ones. It works with ContainerTokenSecretManager (described below) to authorize all requests. All the operations performed on containers running on this node are written to an audit-log which can be post-processed by security tools.
b. **ResourceLocalizationService**: Responsible for securely downloading and organizing various file resources needed by containers. It tries its best to distribute the files across all the available disks. It also enforces access control restrictions of the downloaded files and puts appropriate usage limits on them.
c. **ContainersLauncher**: Maintains a pool of threads to prepare and launch containers as quickly as possible. Also cleans up the containers' processes when such a request is sent by the RM or the ApplicationMasters (AMs).
d. **AuxServices**: The NM provides a framework for extending its functionality by configuring auxiliary services. This allows per-node custom services that specific

frameworks may require, and still sandbox them from the rest of the NM. These services have to be configured before NM starts. Auxiliary services are notified when an application's first container starts on the node, and when the application is considered to be complete.

e. **ContainersMonitor**: After a container is launched, this component starts observing its resource utilization while the container is running. To enforce isolation and fair sharing of resources like memory, each container is allocated some amount of such a resource by the RM. The ContainersMonitor monitors each container's usage continuously and if a container exceeds its allocation, it signals the container to be killed. This is done to prevent any runaway container from adversely affecting other well-behaved containers running on the same node.
f. **LogHandler**: A pluggable component with the option of either keeping the containers' logs on the local disks or zipping them together and uploading them onto a file-system.

2. **ContainerExecutor**

Interacts with the underlying operating system to securely place files and directories needed by containers and subsequently to launch and clean up processes corresponding to containers in a secure manner.

1. **NodeHealthCheckerService**

Provides functionality of checking the health of the node by running a configured script frequently. It also monitors the health of the disks specifically by creating temporary files on the disks every so often. Any changes in the health of the system are notified to NodeStatusUpdater (described above) which in turn passes on the information to the RM.

1. **Security**
    1. **ApplicationACLsManager**NM needs to gate the user facing APIs like container-logs' display on the web-UI to be accessible only to authorized users. This component maintains the ACLs lists per application and enforces them whenever such a request is received.
    2. **ContainerTokenSecretManager**: verifies various incoming requests to ensure that all the incoming operations are indeed properly authorized by the RM.
2. **WebServer**

Exposes the list of applications, containers running on the node at a given point of time, node-health related information and the logs produced by the containers.

Spotlight on Key Functionality

1. **Container Launch**

To facilitate container launch, the NM expects to receive detailed information about a container's runtime as part of the container-specifications. This includes the container's command line, environment variables, a list of (file) resources required by the container and any security tokens.

On receiving a container-launch request – the NM first verifies this request, if security is enabled, to authorize the user, correct resources assignment, etc. The NM then performs the following set of steps to launch the container.

1. A local copy of all the specified resources is created (Distributed Cache).
2. Isolated work directories are created for the container, and the local resources are made available in these directories.
3. The launch environment and command line is used to start the actual container.

2. **Log Aggregation**

Handling user-logs has been one of the big pain-points for Hadoop installations in the past. Instead of truncating user-logs, and leaving them on individual nodes like the TaskTracker, the NM addresses the logs' management issue by providing the option to move these logs securely onto a file-system (FS), for e.g. HDFS, after the application completes.

Logs for all the containers belonging to a single Application and that ran on this NM are aggregated and written out to a single (possibly compressed) log file at a configured location in the FS. Users have access to these logs via YARN command line tools, the web-UI or directly from the FS.

1. **How MapReduce shuffle takes advantage of NM's Auxiliary-services**

The Shuffle functionality required to run a MapReduce (MR) application is implemented as an Auxiliary Service. This service starts up a Netty Web Server, and knows how to handle MR specific shuffle requests from Reduce tasks. The MR AM specifies the service id for the shuffle service, along with security tokens that may be required. The NM provides the AM with the port on which the shuffle service is running which is passed onto the Reduce tasks.

Conclusion

In YARN, the NodeManager is primarily limited to managing abstract containers i.e. only processes corresponding to a container and not concerning itself with per-application state management like MapReduce tasks. It also does away with the notion of named slots like map and reduce slots. Because of this clear separation of responsibilities coupled with the modular architecture described above, NM can scale much more easily and its code is much more maintainable.