YARN

Data and Analytics Training

Big Data – Self Study Guide (207)

This study guide serves as a basis for your self-study on the basics of YARN. The study guide gives you an overview on the main topics and presents a number of internal and external resources you should use to get a deeper understanding of YARN, the related terms and technical details.

Please especially use the external resources provided. They are a main part of this study guide and are required to get a good understanding of the discussed topics.

Estimated Completion Time

2 hour (including time for external resources)

Objectives

After completing this study guide, you will be able to understand

- YARN functionality
- How YARN works
- Components of ResourceManager
- Components of NodeManager

Contents

Introduction

MapReduce has undergone a complete overhaul and we now have, what we call, MapReduce 2.0 (MRv2) or YARN.[1]

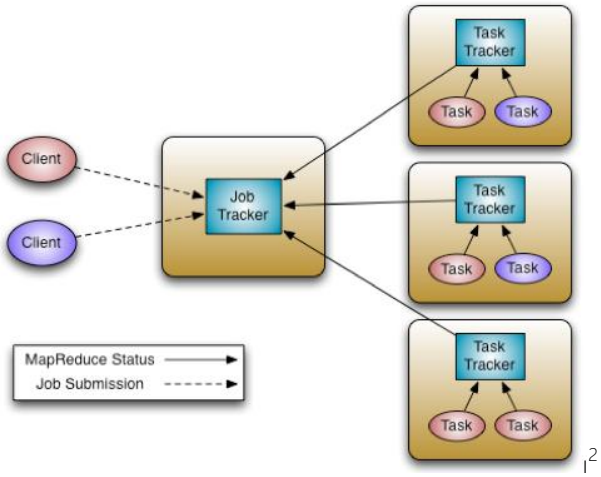
MapReduce 1.0

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.

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.

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



[2]

[1] http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[2] http://hortonworks.com/wp-content/uploads/2012/08/MRArch.png

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, it was understood that the Apache Hadoop MapReduce framework needed an overhaul. In particular, with regards to the JobTracker, it 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.[3]

YARN (MapReduce 2.0)

YARN is the prerequisite for Enterprise Hadoop, providing resource management and a central platform to deliver consistent operations, security, and data governance tools across Hadoop clusters. YARN also extends the power of Hadoop to incumbent and new technologies found within the data center so that they too can take advantage of cost effective, linear-scale storage and processing. It provides ISVs and developers a consistent framework for writing data access applications that run IN Hadoop.[4]

The fundamental idea of YARN is to split up the two major functionalities of the JobTracker, resource management and job scheduling/monitoring, into separate daemons. The idea is to have a global ResourceManager (RM) and per-application ApplicationMaster (AM).

Part of the core Hadoop project, YARN is the architectural center of Hadoop that allows multiple data processing engines such as interactive SQL, real-time streaming, data science and batch processing to handle data stored in a single platform, unlocking an entirely new approach to analytics. It is the foundation of the new generation of Hadoop and is enabling organizations everywhere to realize a modern data architecture.[5]

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.

---

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

[4] http://hortonworks.com/hadoop/yarn/

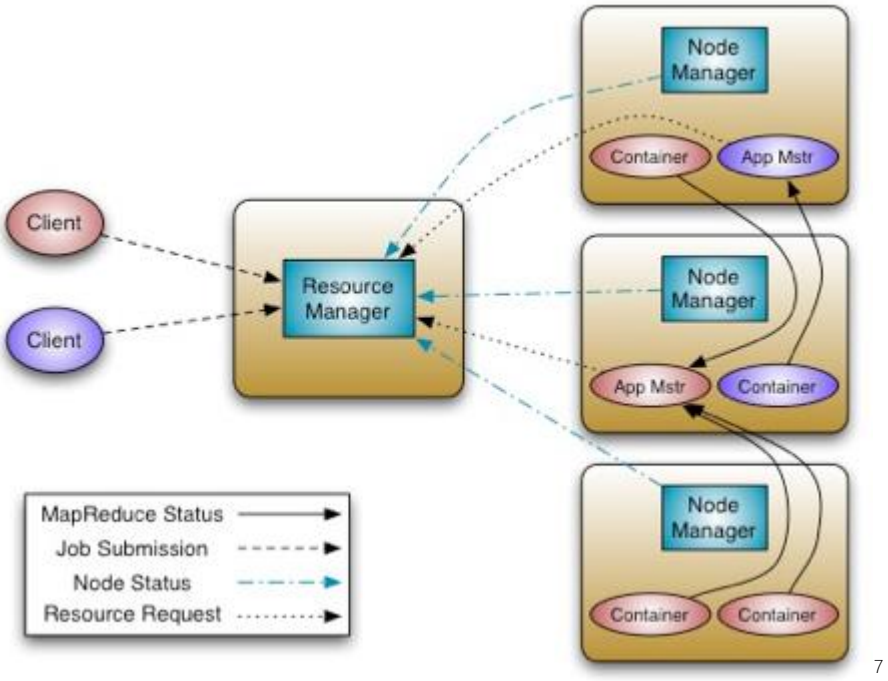
[5] http://hortonworks.com/hadoop/yarn/

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.[6]

Here is an architectural view of YARN:



[7]

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

[7] http://hortonworks.com/wp-content/uploads/2012/08/YARNArch.png

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.

Please watch this video (introduction on YARN is starting at 9:53 minutes):

http://channel9.msdn.com/Events/Build/2014/3-612

Additionally you may want to watch this 47 minutes session recording to get an introduction on YARN:

http://youtu.be/HHv2pkIJjR0

You can find more details on the concepts of YARN here:

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

As its architectural center, YARN enhances a Hadoop compute cluster in the following ways[8]:

- Multi-tenancy: YARN allows multiple access engines (either open-source or proprietary) to use Hadoop as the common standard for batch, interactive and real-time engines that can simultaneously access the same data set. Multi-tenant data processing improves an enterprise's return on its Hadoop investments.
- Cluster untilization: YARN's dynamic allocation of cluster resources improves utilization over more static MapReduce rules used in early versions of Hadoop.
- Scalabiltiy: Data center processing power continues to rapidly expand. YARN's ResourceManager focuses exclusively on scheduling and keeps pace as clusters expand to thousands of nodes managing petabytes of data.
- Compatibility: Existing MapReduce applications developed for Hadoop 1 can run YARN without any disruption to existing processes that already work.

Understanding YARN

YARN's original purpose was to split up the two major responsibilities of the JobTracker/TaskTracker into separate entities:

[8] http://hortonworks.com/hadoop/yarn/

- a global ResourceManager
- a per-application ApplicationMaster
- a per-node slave NodeManager
- a per-application Container running on a NodeManager

The ResourceManager and the NodeManager formed the new generic system for managing applications in a distributed manner. The ResourceManager is the ultimate authority that arbitrates resources among all applications in the system. The ApplicationMaster is a framework-specific entity that negotiates resources from the ResourceManager and works with the NodeManager(s) to execute and monitor the component tasks.

The ResourceManager has a scheduler, which is responsible for allocating resources to the various applications running in the cluster, according to constraints such as queue capacities and user limits. The scheduler schedules based on the resource requirements of each application.

Each ApplicationMaster has responsibility 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.

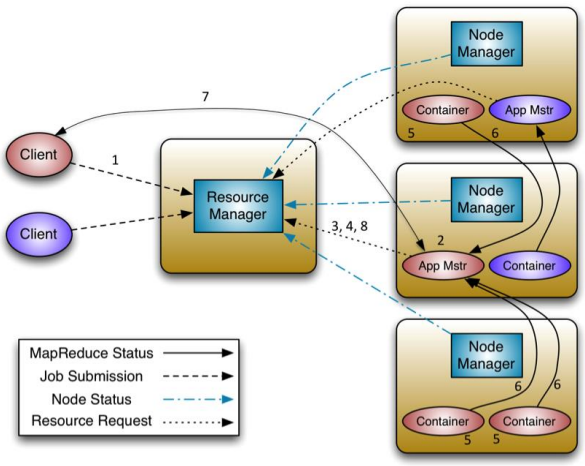
The NodeManager is the per-machine slave, which is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting the same to the ResourceManager.

YARN – Walkthrough

Let's look at how applications conceptually work in YARN. [9]
Application execution consists of the following steps:

1. Application submission.
2. Bootstrapping the ApplicationMaster instance for the application.
3. Application execution managed by the ApplicationMaster instance.

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



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.

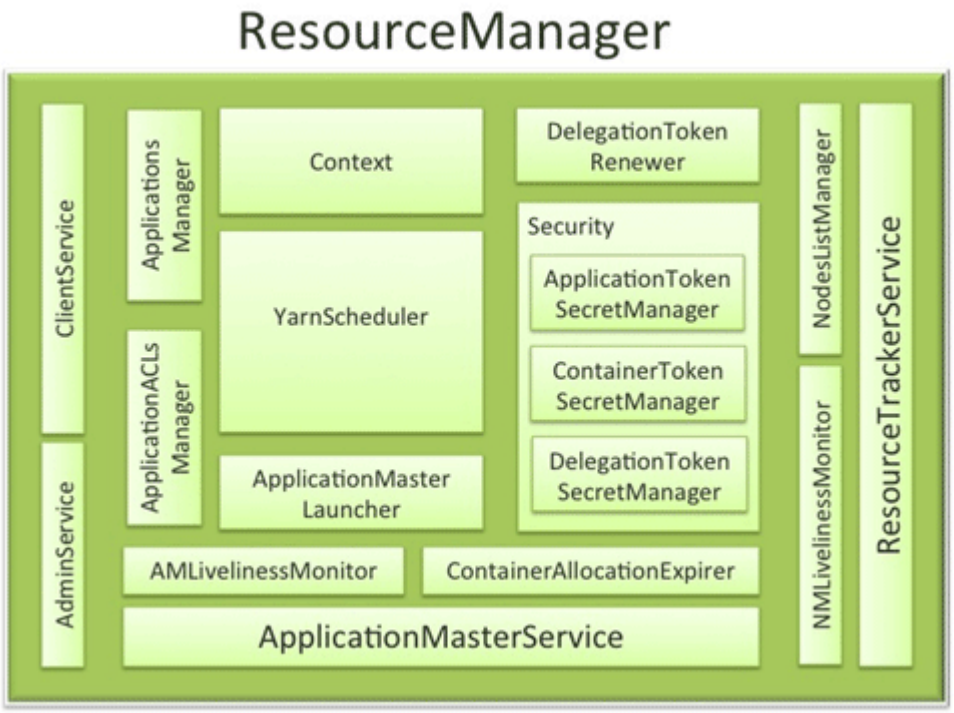
ResourceManager

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).[10]

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):



Components interfacing RM to the clients:

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

- 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.

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.

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.

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 a 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.

- 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.

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.

- 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.

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

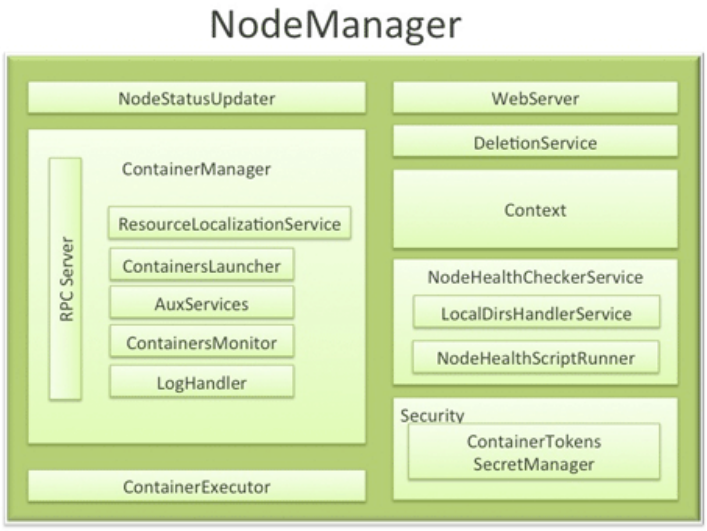
In YARN, the ResourceManager is primarily limited to scheduling i.e. only arbitrating available resources in the system among the competing applications and not concerning itself with per-application state management. Because of this clear separation of responsibilities coupled with the modularity described above, and with the powerful scheduler API, 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.

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.[11]

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



NodeManager Components

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.

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.

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.

2. Security
   A. 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.
   B. ContainerTokenSecretManager: Verifies various incoming requests to ensure that all the incoming operations are indeed properly authorized by the RM.

3. 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.

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.

Application Dependencies

In YARN, applications perform their work by running containers, which today map to processes on the underlying operating system. More often than that, containers have dependencies on files for execution. These files are either required at startup or may be during runtime – just once or more number of times. For example, to launch a simple java program as a container, we need a jar file and potentially more jars as dependencies. Instead of forcing every application to either access (mostly just reading) these files remotely every time or manage the files themselves, YARN gives the applications the ability to localize these files.

At the time of starting a container, an ApplicationMaster (AM) can specify all the files that a container will require and thus should be localized. Once specified, YARN takes care of the localization by itself and hides all the complications involved in securely copying, managing and later deleting these files.