Introducing Apache Hadoop YARN

I'm thrilled to announce that the Apache Hadoop community has decided to promote the next-generation Hadoop data-processing framework, i.e. YARN, to be a sub-project of Apache Hadoop in the ASF!

Apache Hadoop YARN joins Hadoop Common (core libraries), Hadoop HDFS (storage) and Hadoop MapReduce (the MapReduce implementation) as the sub-projects of the Apache Hadoop which, itself, is a Top Level Project in the Apache Software Foundation. Until this milestone, YARN was a part of the Hadoop MapReduce project and now is poised to stand up on it's own as a sub-project of Hadoop.

In a nutshell, Hadoop YARN is an attempt to take Apache Hadoop beyond MapReduce for data-processing.

As folks are aware, Hadoop HDFS is the data storage layer for Hadoop and MapReduce was the data-processing layer. However, the MapReduce algorithm, by itself, isn't sufficient for the very wide variety of use-cases we see Hadoop being employed to solve. With YARN, Hadoop now has a generic resource-management and distributed application framework, where by, one can implement multiple data processing applications customized for the task at hand. Hadoop MapReduce is now one such application for YARN and I see several others given my vantage point - in future you will see MPI, graph-processing, simple services etc.; all co-existing with MapReduce applications in a Hadoop YARN cluster.

Implications for the Apache Hadoop Developer community

I'd like to take a brief moment to walk folks through the implications of making Hadoop YARN as a sub-project, particularly for members of the Hadoop developer community.

> We will now see a top-level hadoop-yarn-project source folder in Hadoop trunk.
> We will now use a separate jira project for issue tracking for YARN i.e. https://issues.apache.org/jira/browse/YARN
> We will also use a new yarn-dev@hadoop.apache.org mailing list for collaboration.

> We will continue to co-release a single Apache Hadoop release that will include the Common, HDFS, YARN and MapReduce sub-projects.

If you would like to play with YARN please download the latest hadoop-2 release from the ASF and start contributing - either to core YARN sub-project or start building your cool application on top!

Please do remember that hadoop-2 is still deemed alpha quality by the Apache Hadoop community, but YARN itself shows a lot of promise and we are excited by the future possibilities!

Conclusion

Overall, having Hadoop YARN as a sub-project of Apache Hadoop is a significant milestone for Hadoop several years in the making. Personally, it is very exciting given that this journey started more than 4 years ago withhttps://issues.apache.org/jira/browse/MAPREDUCE-279. It's a great pleasure, and honor, to get to this point by collaborating with a fantastic community that is driving Apache Hadoop.

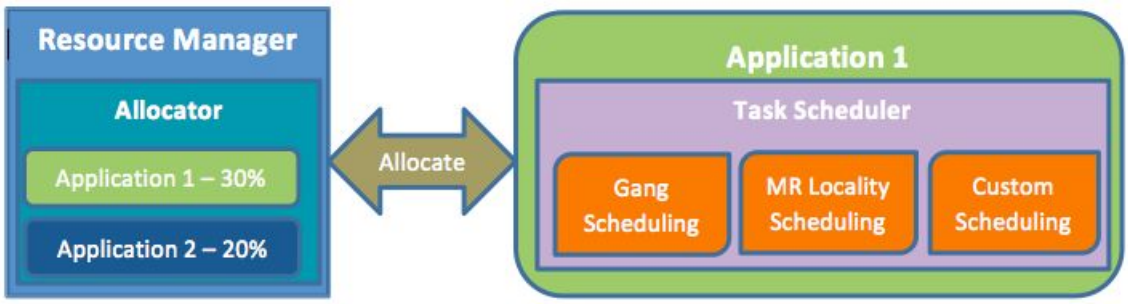
Philosophy behind YARN Resource Management

YARN is part of the next generation Hadoop cluster compute environment. It creates a generic and flexible resource management framework to administer the compute resources in a Hadoop cluster. The YARN application framework allows multiple applications to negotiate resources for themselves and perform their application specific computations on a shared cluster. Thus, resource allocation lies at the heart of YARN.

YARN ultimately opens up Hadoop to additional compute frameworks, like Tez, so that an application can optimize compute for their specific requirements.

The YARN Resource Manager service is the central controlling authority for resource management and makes allocation decisions. It exposes a Scheduler API that is specifically designed to negotiate resources and not schedule tasks. Applications can request resources at different layers of the cluster topology such as nodes, racks etc. The scheduler determines how much and where to allocate based on resource availability and the configured sharing policy.

Currently, there are two sharing policies - fair scheduling and capacity scheduling. Thus, the API reflects the Resource Manager's role as the resource allocator. This API design is also crucial for Resource Manager scalability because it limits the complexity of the operations to the size of the cluster and not the size of the tasks running on the cluster. The actual task scheduling decisions are delegated to the application manager that runs the application logic. It decides when, where and how many tasks to run within the resources allocated to it. It has the flexibility to choose its locality, co-scheduling, co-location and other scheduling strategies.



Fundamentally, YARN resource scheduling is a 2-step framework with resource allocation done by YARN and task scheduling done by the application. This allows YARN to be a generic compute platform while still allowing flexibility of scheduling strategies. An analogy would be general purpose operating systems that allocate computer resources among concurrent processes.

We envision YARN to be the cluster operating system. It may be the case that this 2-step approach is slower than a custom scheduling logic but we believe that such problems can be alleviated by careful design and engineering. Having the custom scheduling logic reside inside the application allows the application to be run on any YARN cluster. This is important for creating a vibrant YARN application ecosystem (tez is a good example of this) that can be easily deployed on any YARN cluster. Developing YARN scheduling libraries will alleviate the developer effort needed to create application specific schedulers and YARN-103 is a step in that direction.

Apache Hadoop YARN – Background & Overview

Celebrating the significant milestone that was Apache Hadoop YARN being promoted to a full-fledged sub-projectof 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:

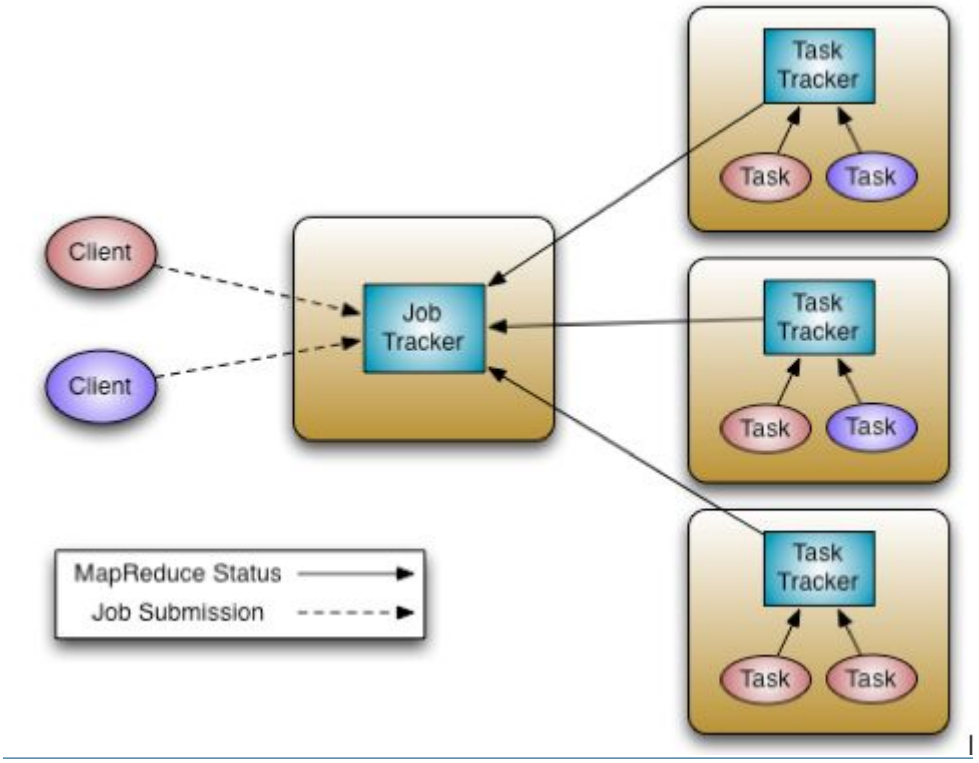
The end-user **MapReduce API** for programming the desired MapReduce application.

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.

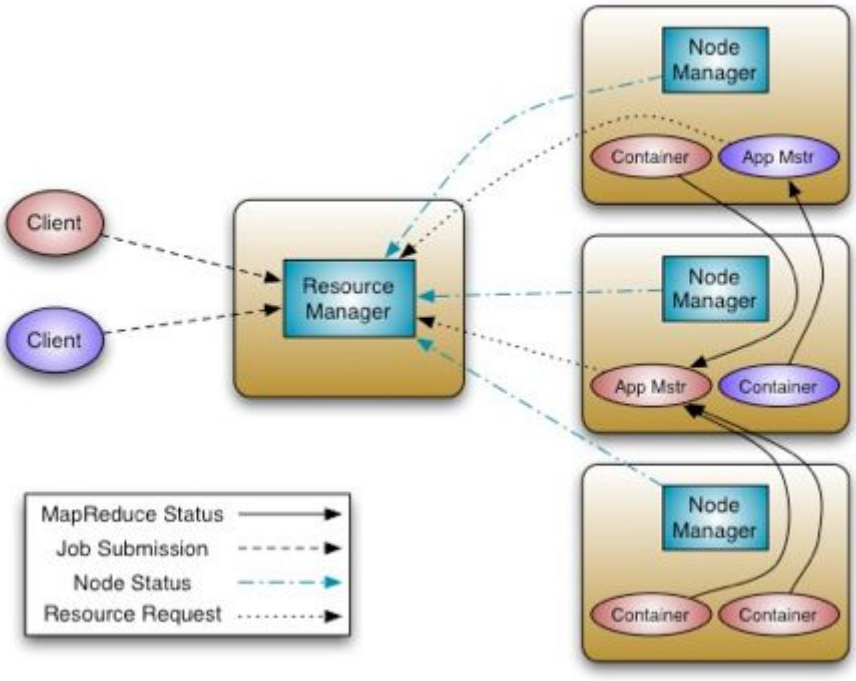
The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application ApplicationMaster is, in effect, a framework specific entity and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute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

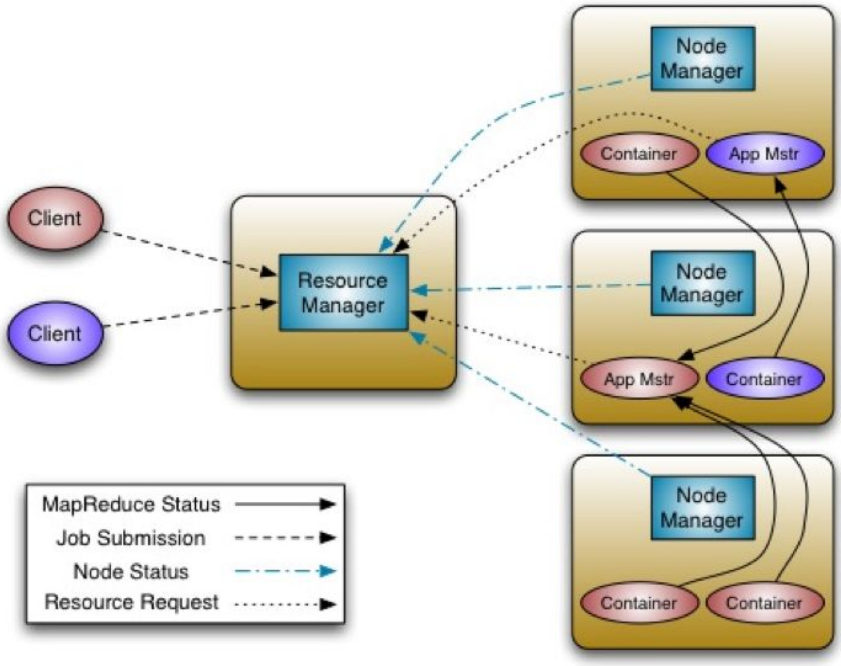


One of the crucial implementation details for MapReduce within the new YARN **system** that I'd like to point out is that we have reused the existing MapReduce **framework** without any major surgery. This was very important to ensure **compatibility** for existing MapReduce applications and users. More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

The next post will dive further into the intricacies of the architecture and its benefits such as significantly better scaling, support for multiple data processing frameworks (MapReduce, MPI etc.) and cluster utilization.

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.

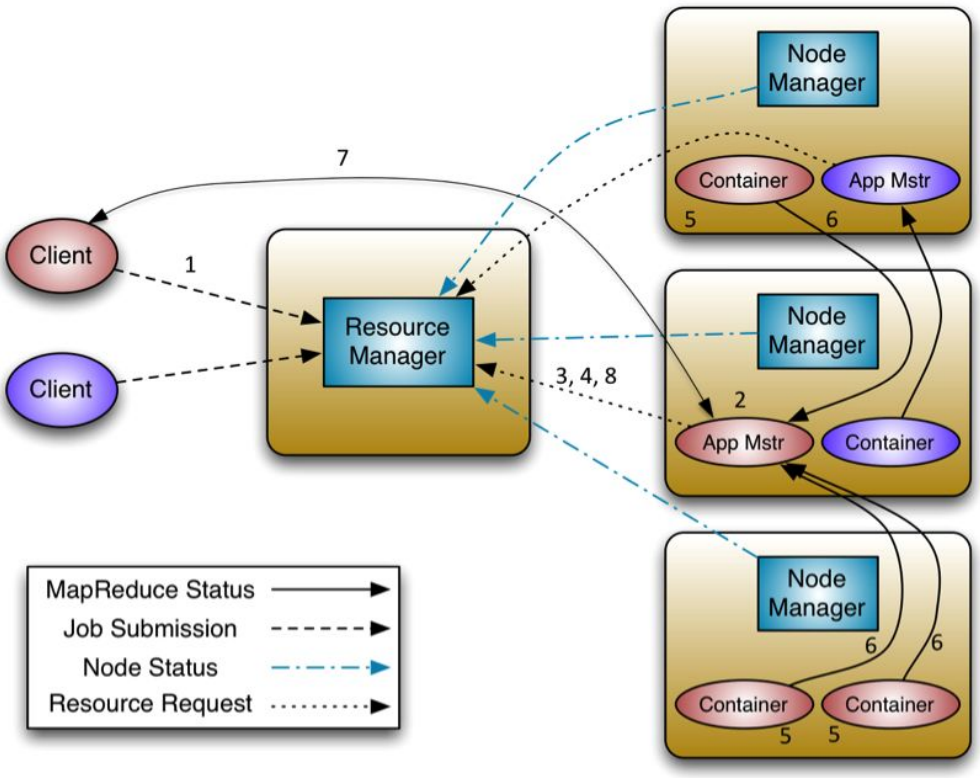
Application execution consists of the following steps:

- 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.

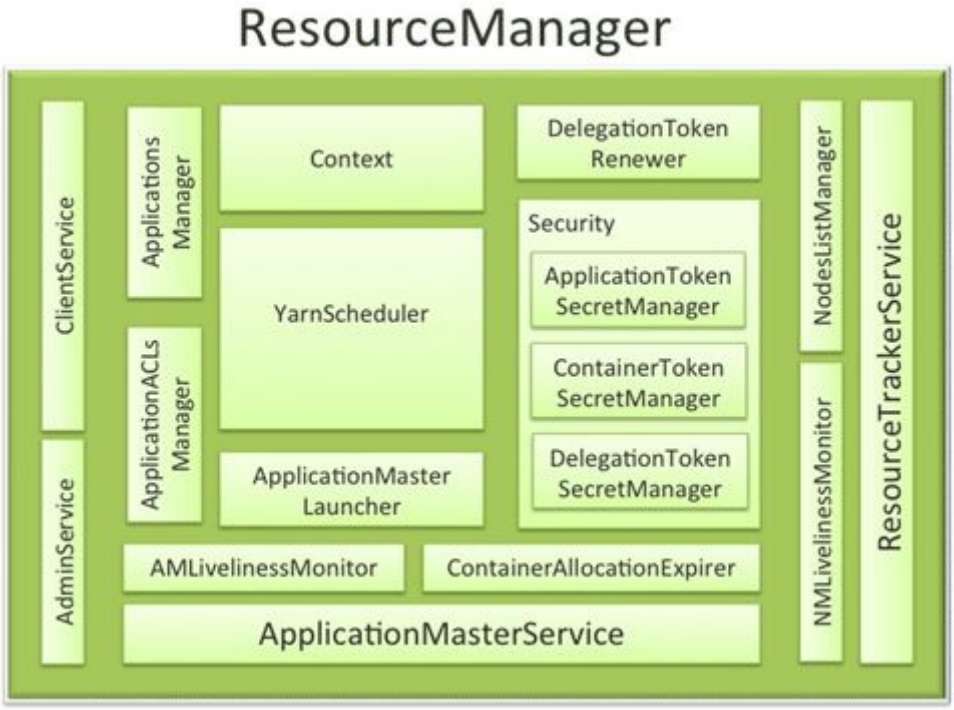


In our next post in this series we dive more into guts of the YARN system, particularly the ResourceManager - stay tuned!

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:**
    1. **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.

2. **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:**

    1. **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.
    2. **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.
    3. **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:**

    1. **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.

2. **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:**

    1. **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.
    2. **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.
    3. **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.
    4. **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.

5. **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:

    1. **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.
    2. **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.
    3. **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.

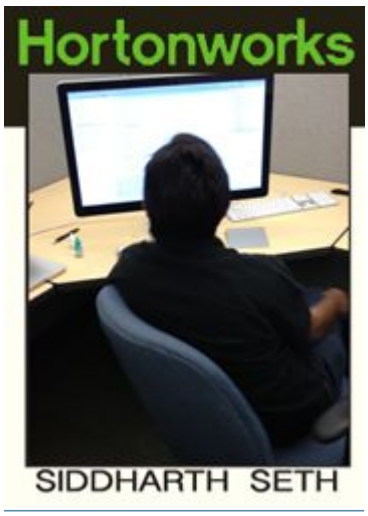
Conclusion

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.

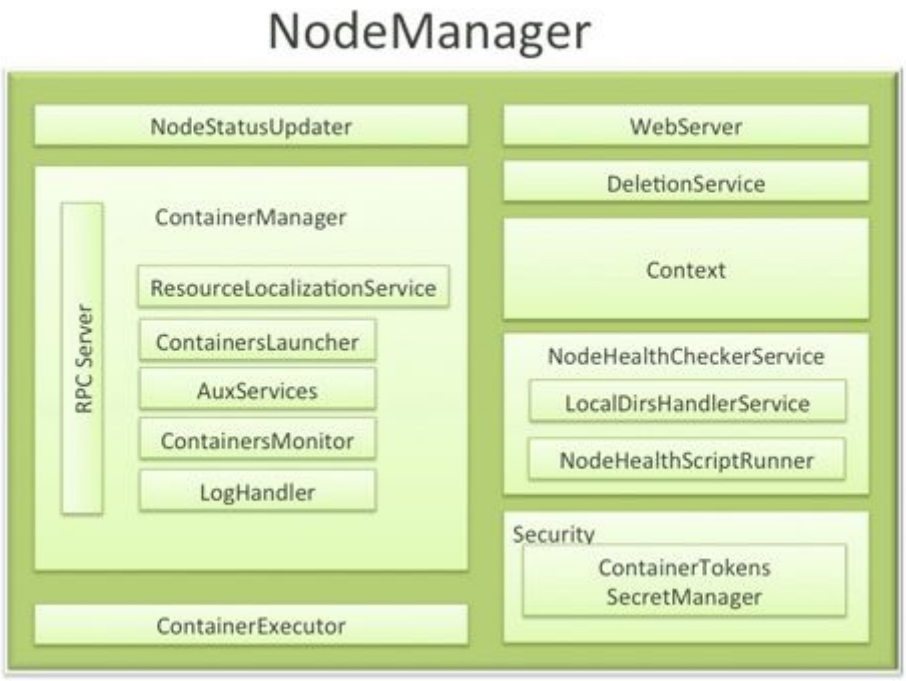
The next post will dive into details of the NodeManager, the component responsible for managing the containers' life cycle and much more.

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.

1. 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.

2. 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.
3. 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).
4. 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.
5. 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.
6. 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. 安全性

    1. ApplicationACLsManagerNM 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.

Running existing applications on Hadoop 2 YARN

Introduction

The beta release of Apache Hadoop 2.x has finally arrived and we are striving hard to make the release easy to adopt with no or minimal pain to our existing users.

As you may know, YARN evolves the compute platform of Hadoop beyond MapReduce so that it can accommodate new processing models, such as streaming and graph processing. To accommodate this transition, a major goal of this release was to ensure that the existing MapReduce applications which were programmed and compiled against previous MapReduce APIs (we'll call these MRv1 applications) can continue to run with little work on top of YARN (calling these as MRv2 applications).

We spent significant amount of time and effort to fix any inadvertently broken MapReduce APIs, so that the upgrade process for old MapReduce applications over to YARN can happen smoothly. Most of this work can be tracked at the MAPREDUCE-5108 JIRA ticket. This post will discuss the backward compatibility with existing MRv1 applications and early MRv2 applications (in particular, those compiled against Hadoop 0.23) that users have written, with the examples that are shipped as part of Hadoop releases. We'll also talk about the compatibility of applications written on top of other frameworks like Pig, Hive, Oozie etc.

Backward compatibility of MRv2 APIs

(1) Binary Compatibility of org.apache.hadoop.mapred APIs

For the vast majority of users who use the the `org.apache.hadoop.mapred` APIs, you have to follow the three steps below:

1.

2.

3.

You read that right, you don't have to do anything! We ensure full binary compatibility! Therefore, these applications can run on YARN directly without recompilation.

You can use jars of your existing application that codes against mapred APIs, and use "bin/hadoop" to submit them directly to YARN!

All you really need to do is to point your application to a YARN installation and HADOOP_CONF_DIR to the corresponding configuration directory. In such conf directory, users will find `yarn-site.xml` (contains the configurations for YARN) and `mapred-site.xml` (contains the configuration for MapReduce apps). Please refer to the full list of YARN configuration properties. In contrast, `mapred.job.tracker` in `mapred-site.xml` is no longer necessary. Instead, users need to add:

```
<property>
<name>mapreduce.framework.name</name>
<value>yarn</value>
</property>
```

in `mapred-site.xml` to run MRv1 applications on YARN. We will cover detailed configuration changes needed to be done by operators in one of the next blog posts.

(2) Source Compatibility of org.apache.hadoop.mapreduce APIs

Unfortunately, it proved to be difficult to ensure full binary compatibility to the existing applications that compiled against MRv1 org.apache.hadoop.mapreduce APIs. These APIs have gone through lots of changes. For example, a bunch of classes stopped being abstract classes and changed to interfaces. Therefore, we compromised to only supporting source compatibility for org.apache.hadoop.mapreduce APIs.

Existing application using mapreduce APIs are source compatible and can run on YARN with no changes, recompilation and/or minor updates.

If MRv1 mapreduce based applications fail to run on YARN, users are requested to investigate their source code, checking whether mapreduce APIs are referred to or not. If they are referred to, users have to

recompile their applications against MRv2 jars that are shipped with Hadoop 2.

(3) Compatibility of Command Line Scripts

Most of the command line scripts over from Hadoop 1.x should just work!

The only exception is MRAdmin whose functionality is removed from MRv2 because JobTracker and TaskTracker aren't there anymore. The MRAdmin functionality is now replaced with RMAdmin. The suggested method to invoke MRAdmin (also RMAdmin) is through command line, even though one can directly invoke the APIs. In YARN, when `mradmin` commands are executed, warning messages will appear, and remind users of using new YARN commands (i.e., rmadmin commands). On the other hand, if users' applications programmatically invoke MRAdmin, applications will break when running on top of YARN. We support neither binary nor source compatibility here.

(4) Compatibility Tradeoff between MRv1 and Early MRv2 (0.23.x) applications

Unfortunately, there are some APIs that can be compatible either with MRv1 applications or with early MRv2 applications (in particular, the applications compiled against Hadoop 0.23), but not both. Some of the APIs were exactly the same in both MRv1 and MRv2 except for the return type change in method signatures. Therefore, we were forced to trade off the compatibility between the two.

1. For mapred APIs, we decided to make them be compatible with MRv1 applications, which have a larger user base.
2. For mapreduce APIs, if they don't significantly break Hadoop 0.23 applications, we made the same decision of continuing to be compatible with 0.23 but only source compatible with 1.x.

Below is the list of APIs which are incompatible with Hadoop 0.23. If early Hadoop 2 adopters using 0.23.x used the following methods in their custom routines, they have to modify the code accordingly. For some problematic methods, we provided an alternative method with the same functionality and similar method signature to MRv2 applications.

Problematic Method - `org.apache.hadoop` Incompatible Return Type Alternative

	Change	Method
.util.ProgramDriver#drive	void -> int	run
.mapred.jobcontrol.Job#getMapredJobID	String -> JobID	getMapredJobId
.mapred.TaskReport#getTaskId	String -> TaskID	getTaskID
.mapred.ClusterStatus#UNINITIALIZED_MEMORY_VALUE	long -> int	N/A
.mapreduce.filecache.DistributedCache#getArchiveTimestamps	long[] -> String[]	N/A
.mapreduce.filecache.DistributedCache#getFileTimestamps	long[] -> String[]	N/A
.mapreduce.Job#failTask	void -> boolean	killTask(TaskAttemptID, boolean)
.mapreduce.Job#killTask	void -> boolean	killTask(TaskAttemptID, boolean)
.mapreduce.Job#getTaskCompletionEvents	.mapred.TaskCompletionEvent[] -> .mapreduce.TaskCompletionEvent[]	N/A

Running Typical Applications on YARN

(1) Running MRv1 examples on YARN

Most of the MRv1 examples continue to work on YARN, except they are now present in a newly versioned jar. One exception worth mentioning is that the sleep example that used to be in `hadoop-examples-1.x.x.jar` is no longer in `hadoop-mapreduce-examples-2.x.x.jar` - it got moved into the test jar `hadoop-mapreduce-client-jobclient-2.x.x-tests.jar`.

That exception aside, users may want to directly try `hadoop-examples-1.x.x.jar` on YARN. Running `hadoop -jar hadoop-examples-1.x.x.jar` will still pick the classes in `hadoop-mapreduce-examples-2.x.x.jar`. This is because by default Java first searches the desired class in the system jars, and if the class is not found, it will go on to search in the user jars in the classpath. `hadoop-mapreduce-examples-2.x.x.jar` is installed together with other MRv2 jars in the Hadoop classpath, such that the desired class (e.g., WordCount) will be picked from this 2.x.x jar instead. However, it is possible to let Java pick the classes from the jar which is specified after -jar option. Users have two options:

1. Add `HADOOP_USER_CLASSPATH_FIRST=true` and `HADOOP_CLASSPATH=...:hadoop-examples-1.x.x.jar` as environment variables, and add `mapreduce.job.user.classpath.first = true` in `mapred-site.xml`.
2. Remove the 2.x.x jar from the classpath. If it is a multiple-node cluster, the jar needs to be removed from the classpath on all the nodes.

(2) Running Pig scripts on YARN

Pig is one of the two major data process applications in the Hadoop ecosystem, the other being Hive. Because of significant efforts from the Pig community, we are happy to report that Pig scripts of existing users don't need any change! Pig on YARN in Hadoop 0.23 has been supported since 0.10.0 and Pig working with Hadoop 2.x has been supported starting 0.10.1.

Existing Pig scripts that work with Pig 0.10.1 and beyond will work just fine on top of YARN !

However, versions earlier to Pig 0.10.x may not run directly on YARN due to some of the incompatible mapreduce APIs and configuration.

(3) Running Hive queries on YARN

Hive queries of existing users don't need any change to work on top of YARN starting Hive-0.10.0, thanks to the Hive community. Support for Hive to work on YARN in Hadoop 0.23 and 2.x releases has been supported since 0.10.0.

Queries on Hive 0.10.0 and beyond will work without changes on top of YARN !

However, like Pig, earlier versions of Hive may not run directly on YARN as those Hive releases don't support 0.23 and 2.x.

(4) Running oozie workflows on YARN

Like Pig and Hive, the Apache Oozie community worked to make sure existing oozie workflows run in a completely backwards compatibility manner. Support for Hadoop 0.23 and 2.x is available starting oozie release 3.2.0.

Existing oozie workflows can start taking advantage of YARN in 0.23 and 2.x with Oozie 3.2.0 and above !

Conclusion

This effort was possible thanks to members of the Apache Hadoop MapReduce community. Mayank Bansal, Karthik Kambatla and Robert Kanter are others who contributed with some patches to this effort. Thanks to the committers Arun C Murthy and Alejandro Abdelnur for helping with reviews and commits.

Thanks are also due to various Apache Hadoop ecosystem communities like Pig, Hive and Oozie to come together in porting the frameworks over to Hadoop 2 YARN with minimal disruption to the end users.

We also are happy to report that some of the installations which kick-started testing and validation of apps on Hadoop 2 YARN, few with big clusters and good sized user-base, are only observing zero or minimal impact with regard to porting their existing MapReduce applications! Looking forward to the beta releases and smooth upgrades of all users' apps!

Stabilizing YARN APIs for Apache Hadoop 2 Beta and beyond

Introduction

Apache Hadoop 2 is in beta now . Hadoop 2 beta brings in YARN that evolves the compute platform of Hadoop beyond MapReduce so that it can accommodate new processing models, such as streaming, graph processing etc. Till now, MapReduce has always been the only framework and hence was the only user facing API to program against for data processing using Hadoop. Starting Hadoop 2, we have YARN APIs themselves which are lower level compared to MR, but let developers write powerful frameworks that can run alongside MapReduce applications on the same cluster.

We recently talked about how existing MR, Pig, Hive, Oozie apps can work on top of YARN. In line with that and so many other similar great things happening for helping users move to Hadoop 2, the Apache Hadoop YARN community recently worked hard to stabilize the YARN APIs. We addressed each and every API issue that we wished to finish before they can be confidently deemed stable.

We engaged various users of our alpha releases, discussed their pain points. We also took feedback from various users and application developers during the Hadoop YARN Meetups (Meetup I and Meetup II). Completion of this stabilization effort now enables us to support stable and apt APIs for a long time and avoiding the potential pain of supporting bad APIs going forward into the beta and stable releases. YARN-386 is the umbrella JIRA issue that tracked this herculean effort.

YARN API changes: Guide for Hadoop 2 alpha users to port apps to Hadoop 2 beta and beyond

We appreciate the efforts of the early adopters (0.23.x and Hadoop-2 alpha users) trying our software and helping to iron out various kinks! In order to smoothen the upgrade process of the users of our alpha releases, we are writing this document to provide information about various YARN API

incompatible changes we introduced when moving from Hadoop 2.0.*-alpha releases to Hadoop 2.1-beta release. We've categorized the changes into three types: (1) Simple renames/replacements (2) Unstable APIs that are completely removed or moved and (3) Miscellaneous changes of note. Detailed changes in each category follow:

1. Below is a list of API methods or classes that have been renamed or replaced.

API/Class/Method	Change and Description
FROM: `YarnException` TO: `YarnRuntimeException`	Renamed. This is a private exception used in YARN and MapReduce. Users aren't supposed to use this.
FROM: `YarnRemoteException` TO: `YarnException`	YarnException indicates exceptions from yarn servers. On the other hand, IOExceptions indicates exceptions from RPC layer either on the server side or the client side.
FROM: `AllocateResponse#(get,set)reboot Methods` TO: `AllocateResponse#(get,set)AMCommand`	Renamed and changed boolean to an enum. AMCommand is a enum and includes AM_RESYNC and AM_SHUTDOWN. It is sent by ResourceManager to ApplicationMaster as part of the AllocateResponse record.
FROM: `ContainerLaunchContext#(get,set)ContainerTokens Methods` TO: `ContainerLaunchContext#(get,set)Tokens`	Renamed. The tokens may include file system tokens, ApplicationMaster related tokens, or framework level tokens needed by this container to communicate to various services in a secure manner.
FROM: `ResourceRequest#(get,set)HostName Methods` TO: `ResourceRequest#(get,set)ResourceName`	Renamed. The resource name on which the allocation is desired. It can be host, rack, or *
FROM: `FinishApplicationMasterRequest#setFinishApplication Status` TO: `FinishApplicationMasterRequest#setFinalApplicationS tatus`	Renamed. The final application-status reported by the ApplicationMaster to the ResourceManager.
FROM: `AMRMClient(Async)#getClusterAvailableResources` :	Renamed. ApplicationMasters can use this method to obtain the

TO: AMRMClient(Async)#getAvailableResources	available resources in the cluster.
FROM YarnClient#getNewApplication : TO: YarnClient#createApplication	Renamed and the return type also has been changed. This provides a directly usable ApplicationSubmissionContext that clients can then use to submit an application.
FROM ApplicationTokenSelector and ApplicationTokenIdentifier : TO: AMRMTokenSelector and AMRMTokenIdentifier	Renamed. The token-selector and the identifier used by ApplicationMaster to authenticate with ResourceManager
FROM ClientTokenSelector and ClientTokenIdentifier : TO: ClientToAMTokenSelector and ClientToAMTokenIdentifier	Renamed. The token-selector and the identifier to be used by clients to authenticate with ApplicationMaster
FROM RMTokenSelector : TO: RMDelegationTokenSelector	Renamed. The selector for RMDelegationToken
FROM RMAdmin : TO: RMAdminCLI	Renamed and changed package. The command line interface to execute Map-Reduce administrative commands. This is a private class that isn't intended to be used by the users directly.
FROM ClientRMProtocol : TO: ApplicationClientProtocol	Renamed. The protocol between client and the ResourceManager to submit/abort jobs and to get information on applications, nodes, queues, and ACLs.
FROM AMRMProtocol : TO: ApplicationMasterProtocol	Renamed. The protocol between a live ApplicationMaster and ResourceManager to register/unregister ApplicationMaster and request resources in the cluster from ResourceManager
FROM ContainerManager : TO: ContainerManagementProtocol	Renamed. The protocol between ApplicationMaster and a NodeManager to start/stop containers and to get status of

	running containers.
FROM `RMAdminProtocol` : TO: `ResourceManagerAdministrationProtocol`	Renamed and changed package from org.apache.hadoop.yarn.api to org.apache.hadoop.yarn.server.api
FROM `yarn.app.mapreduce.container.log.dir` : TO: `yarn.app.container.log.dir`	Configuration property moved from Mapreduce into YARN and renamed. Represents the log directory for the containers if the AM uses the generic container-log4j.properties.
FROM `yarn.app.mapreduce.container.log.filesize` : TO: `yarn.app.container.log.filesize`	Configuration property moved from Mapreduce into YARN and renamed.
FROM `ApplicationClientProtocol#getAllApplications` : TO: `ApplicationClientProtocol#getApplications`	Renamed and changed to accept a list of ApplicationTypes as a parameter with which to filter the applications
FROM `ApplicationClientProtocol#getClusterNodes` :	Changed to accept a list of node states with which to filter the cluster nodes and to be consistent with web-services related to nodes.
FROM `ContainerManagementProtcol` :	All APIs are changed to take in requests for multiple containers

2. API methods or classes that are either removed or moved out.

API/Record	Moved or removed?
`BuilderUtils`	Moved and made it YARN private. User should instead use record specific static factory method to construct new records.
`AMResponse`	AMResponse is merged into AllocateResponse. Use AllocateResponse to retrieve all responses sent by ResourceManager to ApplicationMaster during resource negotiation.
`ClientToken, DelegationToken, ContainerToken`	Removed. Instead, use the org.apache.hadoop.yarn.api.records.Token as the common type for ClientToAMToken, DelegationToken and ContainerToken.

Container#(get,set)ContainerState (get,set)ContainerStatus	Removed from Container as they were always unusable inside Container.
ContainerExitStatus	Removed from YarnConfiguration and become as a separate API record.
ContainerLaunchContext#(get,set)User	Removed. User-name is already available in ContainerTokenIdentifier in ContainerToken which is passed as part of StartContainerRequest.
(get,set)MinimumResourceCapability	Removed from RegisterApplicationMasterResponse and GetNewApplicationResponse. These two methods are supposed to be internal to the scheduler.
APPLICATION_CLIENT_SECRET_ENV_NAME	Removed from ApplicationConstants. It is now sent in RegisterApplicationMasterResponse by ResourceManager to a new ApplicationMaster on registration, instead of sharing it by setting into the environment of the Containers.
APPLICATION_MASTER_TOKEN_ENV_NAME	Removed from ApplicationConstants. AMRMTokens are now available in the token-file of the AM and can be access from the current UserGroupInformation.
RegisterApplicationMasterRequest#(get,set)ApplicationAttemptId	Removed. It's not needed any more as AMRMToken is now changed to be used irrespective of secure or non-secure environment

3. More changes of note

The YARN protocols are sometimes too low level to program against and so we added a bunch of client libraries in yarn-client module. These libraries can help enhance developer productivity by taking care of some of the boiler plate code. Specifically, we added via YARN-418

- YarnClient : For all communications from the client to the ResourceManager.
- AMRMClient : For ApplicationMasters to communicate to ResourceManager for requesting resources, registering etc. There are two types of clients here - >AMRMClient for

blocking calls and AMRMClientAsync for non-blocking calls. We strongly encourage users to take advantage of the async APIs

- NMClient : For ApplicationMasters to talk to NodeManager for launching, monitoring and stopping containers on the NodeManagers. Even here, there are two types of clients - NMClient for blocking calls to a single NM and NMClientAsync for non-blocking calls to any NM in the cluster. NMClientAsync is the suggested library as it also packs other useful features like thread management for connections to any or all the nodes in the cluster.

ContainerManagementProtocol#startContainers is changed to accept ContainerToken for each container as a parameter so that ContainerLaunchContext is completely user land. ContainerLaunchContext only needs information that has to be set by client/ApplicationMaster, everything else like ContainerToken is taken care of transparently via the Container record.

Concept of a separate NMToken and ContainerToken : NMTokens are now used for authenticating all the communication with NodeManager. It is issued by ResourceManager when ApplicationMaster negotiates resource with ResourceManager and is validated on the NodeManager side.

- ContainerToken is now only used for authorization and only during starting of a container to make sure that the user(identified by application-submitter) is valid, or token is not expired.

NMTokens are shared between AMRMClient and NMClient using NMTokenCache (api-based) instead of a memory-based approach. NMTokenCache is a static token cache which will be created one per AM. AMRMClient puts newly received NMTokens in it and NMClient can pick up NMTokens from there to create authenticated connection with NodeManager.

Every container launched by NodeManager now contains some key information in its environment such as containerId, container log directory, NM hostname, NM port. See ApplicationConstants.java for more information.

All protocol APIs (`ApplicationClientProtocol` etc.) are changed to throw two exceptions: (1)`YarnException` which indicates exceptions from yarn servers and (2) `IOException` which indicates exceptions from RPC layer.

All IDs (`ApplicationId, ContainerId` etc.) are made immutable. User can not modify the IDs after the IDs are constructed.

`AuxiliaryService` which allows per-node custom services has become part of yarn user-facing API. This is a generic service that is started by `NodeManager` for extending its functionality that administrators have to configure on each node by setting `YarnConfiguration#NM_AUX_SERVICES`

`AMRMToken` is used irrespective of secure or non-secure environment.

Acknowledgements

These API changes done by the community enable us to confidently support direct users of YARN APIs, essentially framework developers, for a long long time. We'd like to shout out names of all the contributors who helped us make this gigantic leap towards stability. Zhijie Shen, Omkar Vinit Joshi, Xuan Gong, Sandy Ryza, Hitesh Shah, Siddharth Seth, Bikas Saha, Arun C Murthy, Alejandro Abdelnur, among many others, have contributed to this huge effort. Thanks everyone and happy porting!

Management of Application Dependencies in YARN

Introduction

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.

In the remainder of this post, we'll explain the basic concepts about this functionality.

LocalResources: Definitions

Here are some definitions to begin with:

Localization. Localization is the process of copying/download remote resources onto the local file-system. Instead of always accessing a resource remotely, it is copied to the local machine which can then be accessed locally.

LocalResource. LocalResource represents a file/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:

URL: Remote location from where a LocalResource has to be downloaded
Size: Size in bytes of the LocalResource
Last-modification **timestamp** of the resource on the remote file-system before container-start.
LocalResourceType: Specifies the type of a resource localized by the NodeManager - FILE, ARCHIVE and PATTERN
Pattern: the pattern that should be used to extract entries from the archive (only used when type is PATTERN).
LocalResourceVisibility: Specifies the visibility of a resource localized by the NodeManager. The visibility can be one of PUBLIC, PRIVATE and APPLICATION

What files can a container request for localization? One can use any kind of files that are meant to be read-only by the containers. Typical examples of LocalResources include:

- Libraries required for starting the container such as a jar file
- Configuration files required to configure the container once started (remote service urls, application default configs etc).
- A static dictionary file.

The following are some examples of bad candidates for LocalResources:

- Shared files that external components may potentially update in future and current containers wish to track these changes,
- Files that applications themselves directly want to update or
- File through which an application plans to share the updated information with external services.

Other related definitions

- **ResourceLocalizationService**: As previously described in the post about NodeManager, ResourceLocalizationService is the service inside NodeManager that is 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, enforces access control restrictions of the downloaded files and puts appropriate usage limits on them.
- **DeletionService**: A service that runs inside the NodeManager and deletes local paths as and when instructed to do so.

- Localizer: The actual thread or process that does Localization. There are two types of Localizers – PublicLocalizer for PUBLIC resources and ContainerLocalizers for PRIVATE and APPLICATION resources.
- LocalCache: NodeManager maintains and manages serveral local-cache of all the files downloaded. The resources are uniquely identified based on the remote-url originally used while copying that file.

LocalResource time-stamps

As mentioned above, NodeManager tracks the last-modification timestamp of each LocalResource before container-start. Before downloading, NodeManager checks that the files haven't changed in the interim. This is a way of giving 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 and hence if an external system has to update the remote resource – it should be done via versioning. YARN will fail containers that depend on modified remote resources to prevent inconsistencies.

Note that ApplicationMaster specifies the resource time-stamps to a NodeManager while starting any container on that node. Similarly, for the container running the ApplicationMaster itself, the client has to populate the time-stamps 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 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 jars, tars, tar.gz files 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-filesystem during localizaiton. Both the original file and extracted files are put in the same directory. Which contents have to extracted from the ARCHIVE and which shouldn't be is 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 Visibilites

LocalResources can be of three types depending upon their specified LocalResourceVisibility, i.e 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 public LocalCache. If in 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 thus not copied/downloaded again if it isn't evicted from the LocalCache by then. All files in public cache will be owned by "yarn-user" (user which 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 LocalResourcse ar ecopied into the specific user's (user who started the container i.e. the application submitter's) private cache. These files are accessible to all the

containers belonging to different applications but all started by the same user. These files on local file system are owned by the user and not accessible by any other user. Similar to public LocalCache, even for the application submitters, there aren't any write permissions - the user cannot modify these files once localized. This is to avoid accidental writes to these files by one container harming other containers - all containers expect them to be in the same state as originally specified (mirroring original timestamp and/or version number).

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

Notes on LocalResource Visibilities
Note that ApplicationMaster specifies this resource visibility to a NodeManager while starting the container - Node manager itself doesn't make any decision and classify resources. Similarly, for the container running the ApplicationMaster itself, the client has to specify visibilities for all the resources that ApplicationMaster needs.

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

Life-time of the LocalResources

Like already mentioned, different type of LocalResources have different life-cycles:

- PUBLIC LocalResources are not deleted once the container or application finishes. They are only deleted when there is a 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 described below.

- **PRIVATE** LocalResources also follow the same life-cycle as PUBLIC resources. In future, we wish to track separate thresholds for different users.
- **APPLICATION** scoped LocalResources are deleted immediately after the application finishes.

One thing of note is that for any given application, we may have multiple ApplicationAttempts and each attempt may start zero or more containers on a given node manager. When the first container belonging to an ApplicationAttempt starts, ResourceLocalizationService localizes files for that application as requested in the container's launch context. If future containers request more such resources then they all will be localized. If one ApplicationAttempt finishes/fails and another is started, ResourceLocalizationService doesn't do anything w.r.t the previously localized resources. However when eventually the application finishes, 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.

Conclusion

That ends our coverage of the basic concepts that application writers will need to know about LocalResources. LocalResources are a very useful feature that application writers can exploit to declare their startup and runtime dependencies. In the next post, we'll delve deep into how the localization process itself actually takes place in the NodeManager.

Resource Localization in YARN: Deep Dive

Introduction

In the previous post, we explained the basic concepts of LocalResources and resource localization in YARN. In this post, we'll dig deeper into the innards explaining how the localization happens inside NodeManager.

Recap of definitions

A brief recap of some definitions follows.

Localization: Localization is the process of copying/download remote resources onto the local file-system. Instead of always accessing a resource remotely, it is copied to the local machine which can then be accessed locally.

LocalResource: LocalResource represents a file/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

- **URL**: Remote location from where a LocalResource has to be downloaded
- **Size**: Size in bytes of the LocalResource
- Creation **timestamp** of the resource on the remote file-system
- **LocalResourceType**: Specifies the type of a resource localized by the NodeManager - FILE, ARCHIVE and PATTERN
- **Pattern**: the pattern that should be used to extract entries from the archive (only used when type is PATTERN).
- **LocalResourceVisibility**: Specifies the visibility of a resource localized by the NodeManager. The visibility can be one of PUBLIC, PRIVATE and APPLICATION

ResourceLocalizationService: The service inside NodeManager that is responsible for localization.

DeletionService: A service that runs inside the NodeManager and deletes local paths as and when instructed to do so.

Localizer: The actual thread or process that does Localization. There are two types of Localizers – PublicLocalizer for PUBLIC resources and ContainerLocalizers for PRIVATE and APPLICATION resources.

LocalCache: NodeManager maintains and manages serveral local-cache of all the files downloaded. The resources are uniquely identified based on the remote-url originally used while copying that file.

How localization works

As you recall from the previous post, there are three types of LocalResources – PUBLIC, PRIVATE and APPLICATION specific resources. PUBLIC LocalResources are localized separately by the NodeManager from PRIVATE/APPLICATION LocalResources because of security implications.

Localization of PUBLIC resources

Localization of PUBLIC resources is taken care of by a pool of threads called PublicLocalizers.

- PublicLocalizers run inside the address-space of the NodeManager itself.
- The number of PublicLocalizer threads is controlled by the configuration property**yarn.nodemanager.localizer.fetch.thread-count** – maximum parallelism during downloading of PUBLIC resources is equal to this thread count.
- While localizing PUBLIC resources, the localizer validates that all the requested resources are indeed PUBLIC by checking their permissions on the remote file-system. Any LocalResource that doesn't fit that condition is rejected for localization.
- Each PublicLocalizer uses credentials passed as part of ContainerLaunchContext to securely copy the resources from the remote file-system.

Localization of PRIVATE/APPLICATON resources

Localization of PRIVATE/APPLICATION resources is not done inside the NodeManager and hence is not centralized. The process is a little involved and is outlined below:

- Localization of these resources happen in a separate process called ContainerLocalizer.
- Every ContainerLocalizer process is managed by a single thread in NodeManager called LocalizerRunner. Every container will trigger one LocalizerRunner if it has any resources that are not yet downloaded.
- LocalResourcesTracker is a per-user or per-application object that tracks all the LocalResources for a given user or an application.
- When a container first requests a PRIVATE/APPLICATION LocalResource, if it is not found in LocalResourcesTracker (or found but in INITIALIZED state) then it is added to pending-resources list.
    - A LocalizerRunner may(or may not) get created depending on the need for downloading something new.
    - The LocalResources is added to its LocalizerRunner's pending-resources list.
- One requirement for NodeManager in secure mode is to download/copy these resources as the application-submitter and not as a yarn-user (privileged user). Therefore LocalizerRunner starts a LinuxContainerExecutor(LCE) (a process running as application-submitter) which then execs a ContainerLocalizer to download these resources.
    - Once started, ContainerLocalizer starts heartbeating with the NodeManager process.
    - On each heartbeat, LocalizerRunner either assigns one resource at a time to a ContainerLocalizer or asks it to die. ContainerLocalizer informs LocalizerRunner about the status of the download.
    - If it fails to download a resource, then that particular resource is removed from LocalResourcesTracker and the container eventually is marked as failed. When this happens LocalizerRunners stops the running ContainerLocalizers and exits.
    - If it is a successful download, then LocalizerRunner gives a ContainerLocalizer another resource again and again until all pending resources are successfully downloaded.

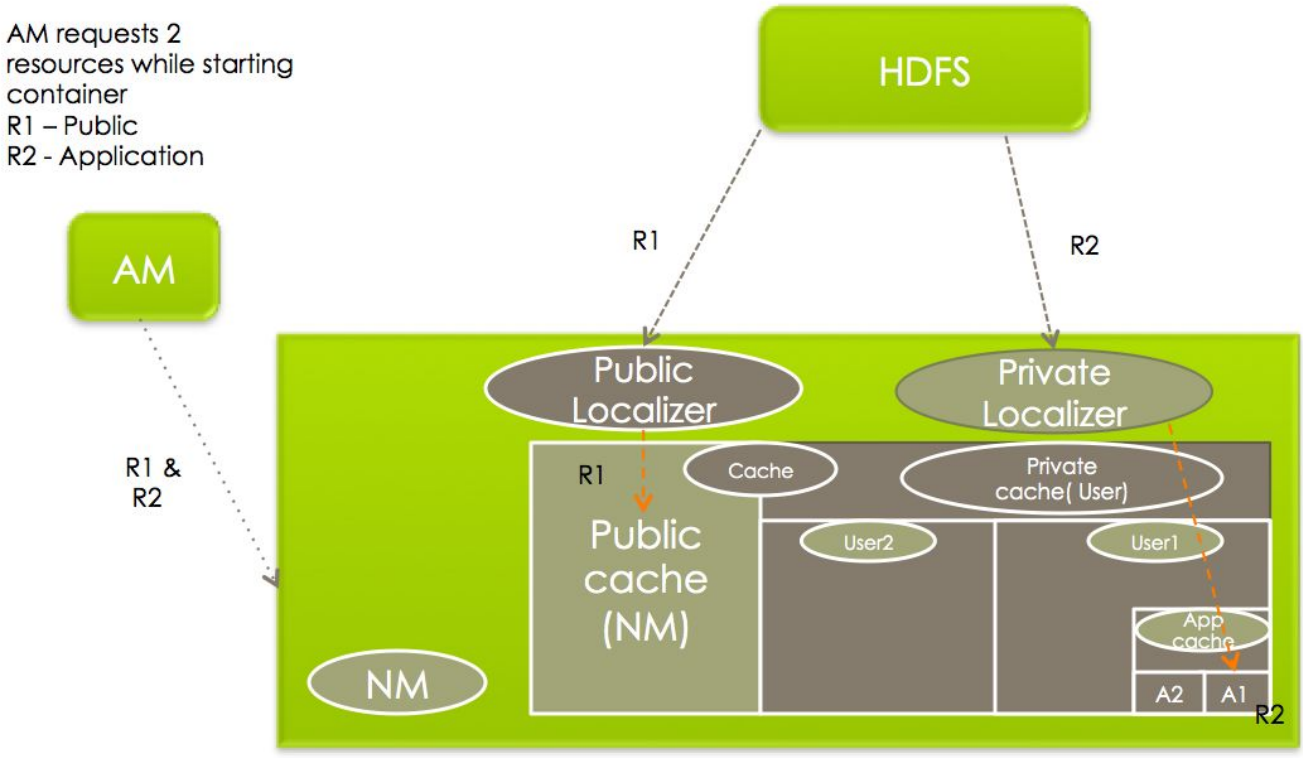
- At present, each ContainerLocalizer doesn't support parallel download of mutliple PRIVATE/APPLICATION resources which we are trying to fix via YARN-574.

Note that because of the above, the maximum parallelism that we can get at present is the number of containers requested for same user on same node manager at THAT point of time. This in the worst case is one when an ApplicationMaster itself is starting. So if AM needs any resources to be localized then today they will be downloaded serially before its container starts.

Target locations of LocalResources

On each of the NodeManager machines, LocalResources are ultimately localized in the following target directories, under each local-directory:

- PUBLIC: `<local-dir>/filecache`
- PRIVATE: `<local-dir>/usercache//filecache`
- APPLICATION: `<local-dir>/usercache//appcache/<app-id>/`



Configuration for resources' localization

Administrators can control various things related to resource-localization by setting or changing certain configuration parameters in yarn-site.xml when starting a NodeManager.

- **yarn.nodemanager.local-dirs**: This is a comma separated list of local-directories that one can configure to be used for copying files during localization. The idea behind allowing multiple directories is to use multiple disks for localization - it helps both fail-over (one/few disk(s) going bad doesn't affect all containers) and load balancing (no single disk is bottlenecked with writes). Thus, individual directories should be configured if possible on different local disks.
- **yarn.nodemanager.local-cache.max-files-per-directory**: Limits the maximum number of files which will be localized in each of the localization directories (separately for PUBLIC / PRIVATE / APPLICATION resources). Its default value is 8192 and should not typically be assigned a large value (configure a value which is sufficiently less than the per directory maximum file limit of the underlying file-system e.g ext3).
- **yarn.nodemanager.localizer.address**: The network address where ResourceLocalizationService listens to for various localizers.
- **yarn.nodemanager.localizer.client.thread-count**: Limits the number of RPC threads in ResourceLocalizationService that are used for handling localization requests from Localizers. Defaults to 5, which means that by default at any point of time, only 5 Localizers will be processed while others wait in the RPC queues.
- **yarn.nodemanager.localizer.fetch.thread-count**: Configures the number of threads used for localizing PUBLIC resources. Recall that localization of PUBLIC resources happens inside the NodeManager address space and thus this property limits how many threads will be spawned inside NodeManager for localization of PUBLIC resources. Defaults to 4.
- **yarn.nodemanager.delete.thread-count**: Controls the number of threads used by DeletionService for deleting files. This DeletionUser is used all over the NodeManager for deleting log files as well as local cache files. Defaults to 4.

- **yarn.nodemanager.localizer.cache.target-size-mb**: This decides the maximum disk space to be used for localizing resources. (At present there is no individual limit for PRIVATE / APPLICATION / PUBLIC cache. YARN-882). Once the total disk size of the cache exceeds this then Deletion service will try to remove files which are not used by any running containers. At present there is no limit (quota) for user cache / public cache / private cache. This limit is applicable to all the disks as a total and is not based on per disk basis.
- **yarn.nodemanager.localizer.cache.cleanup.interval-ms**: After this interval resource localization service will try to delete the unused resources if total cache size exceeds the configured max-size. Unused resources are those resources which are not referenced by any running container. Every time container requests a resource, container is added into the resources' reference list. It will remain there until container finishes avoiding accidental deletion of this resource. As a part of container resource cleanup (when container finishes) container will be removed from resources' reference list. That is why when reference count drops to zero it is an ideal candidate for deletion. The resources will be deleted on LRU basis until current cache size drops below target size.

Conclusion

That concludes our exposition of resource-localization in NodeManager. It is one of the chief services offered by NodeManagers to the applications. Next time, we'll continue with more gritty details of YARN, stay tuned.