Architecture of Next Generation Apache Hadoop MapReduce Framework

Authors:
Arun C. Murthy, Chris Douglas, Mahadev Konar, Owen O'Malley, Sanjay Radia, Sharad Agarwal, Vinod K V

BACKGROUND

The Apache Hadoop Map-Reduce framework is showing it's age, clearly.

In particular, the Map-Reduce JobTracker needs a drastic overhaul to address several technical deficiencies in its memory consumption, much better threading-model and scalability/reliability/performance given observed trends in cluster sizes and workloads. Periodically, we have done running repairs. However, lately these have come at an ever-growing cost as evinced by the worrying regular site-up issues we have seen in the past year. The architectural deficiencies, and corrective measures, are both old and well understood - even as far back as late 2007: https://issues.apache.org/jira/browse/MAPREDUCE-278.

REQUIREMENTS

As we consider ways to improve the Hadoop Map-Reduce framework it is important to keep in mind the high-level requirements. The most pressing requirements for the next generation of the Map-Reduce framework from the customers of the Hadoop, as we peer ahead into 2011 and beyond, are:

- Reliability
- Availability
- Scalability - Clusters of 10000 nodes and 200,000 cores
- Backward Compatibility - Ensure customers' Map-Reduce applications can run unchanged in the next version of the framework. Also implies forward compatibility.
- Evolution – Ability for customers to control upgrades to the grid software stack.
- Predictable Latency – A major customer concern.
- Cluster utilization

The second tier of requirements is:

- Support for alternate programming paradigms to Map-Reduce
- Support for limited, short-lived services

Given the above requirements, it is clear that we need a major re-think of the infrastructure used for data processing on the Grid. There is consensus around the fact that the current architecture of the Map-Reduce framework is incapable of meeting our states goals and that a Two Level Scheduler is the future.

NEXTGEN MAPREDUCE (MRv2/YARN)

The fundamental idea of MRv2 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)**. *An application is either a single job in the classical sense of Map-Reduce jobs or a DAG of jobs*. The ResourceManager and per-node slave, the **NodeManager (NM)**, form the data-computation framework. 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 library and is tasked with negotiating resources from the ResourceManager and working with the NodeManager(s) to execute and monitor the tasks.

The ResourceManager has two main components:

- Scheduler (S)
- ApplicationsManager (ASM)

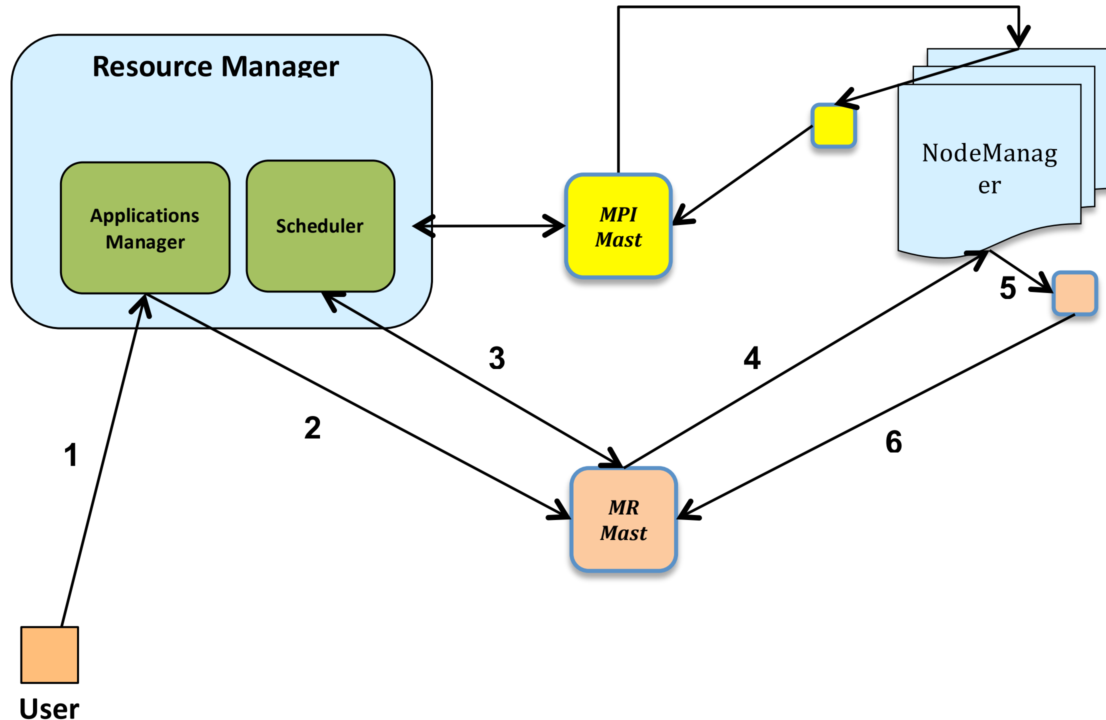
The Scheduler is responsible for allocating resources to the various running applications subject to familiar constraints of capacities, queues etc. The Scheduler is *pure scheduler* in the sense that it performs no monitoring or tracking of status for the application. Also, it offers no guarantees on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

The Scheduler has a pluggable policy plug-in, which is responsible for partitioning the cluster resources among the various queues, applications etc. The current Map-Reduce schedulers such as the CapacityScheduler and the FairScheduler would be some examples of the plug-in.

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure.

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

The per-application ApplicationMaster has the responsibility of negotiating appropriate resource containers from the Scheduler, tracking their status and monitoring for progress.



YARN Architecture

YARN v1.0

This section describes the POR for the first version of YARN.

Requirements for v1

Here are the requirements for YARN-1.0:

- Reliability
- Availability
- Scalability - Clusters of 6000 nodes and 100,000 cores
- Backward Compatibility - Ensure customers' Map-Reduce applications can run. Also implies forward compatibility.
- Evolution – Ability for customers to control upgrades to the grid software stack.
- Cluster utilization

ResourceManager

The ResourceManager is the central authority that arbitrates *resources* among various competing *applications* in the cluster.

The ResourceManager has two main components:

- Scheduler
- ApplicationsManager

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

The ApplicationsManager is responsible for accepting job-submissions, negotiating the first container for executing the application specific ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure.

Resource Model

The Scheduler models only memory in YARN v1.0. Every node in the system is considered to be composed of multiple containers of minimum size of memory (say 512MB or 1 GB). The ApplicationMaster can request any container as a multiple of the minimum memory size.

Eventually we want to move to a more generic resource model, however, for Yarn v1 we propose a rather straightforward model:

The resource model is completely based on memory (RAM) and every node is made up discreet chunks of memory.

Unlike Hadoop Map-Reduce the cluster is not artificially segregated into map and reduce slots. Every chunk of memory is fungible and this has huge benefits for cluster utilization - currently a well known problem with Hadoop Map-Reduce is that jobs are bottlenecked on reduce slots and the lack of fungible resources is a severe limiting factor.

In Yarn an application (via the ApplicationMaster) can ask for containers spanning any number of memory *chunks*, they can ask for varied number of container types also. The ApplicationMaster, usually, asks for specific hosts/racks with specified container capabilities.

Resource Negotiation

The ApplicationMaster can request for containers with appropriate resource requirements, inclusive of specific machines. They can also request for multiple containers on each machine. All resource requests are subject to capacity constraints for the application, it's queue etc.

The ApplicationMaster is responsible for computing the resource requirements of the application e.g. *input-splits* for MR applications, and translating them into the protocol understood by the Scheduler. The protocol understood by the Scheduler is *<priority, (host, rack, *), memory, #containers>*.

In the Map-Reduce case, the Map-Reduce ApplicationMaster takes the input-splits and presents to the ResourceManager Scheduler an inverted table keyed on the hosts, with limits on total containers it needs in it's life-time, subject to change.

A typical resource request from the per-application ApplicationMaster:

Priority	Hostname	Resource Requirements (memory in GB)	Number of Containers
1	h1001.company.com	1GB	5
1	h1010.company.com	1GB	3
1	h2031.company.com	1GB	6
1	rack11	1GB	8
1	rack45	1GB	6
1	*	1GB	14
2	*	2GB	3

The Scheduler will try to match appropriate machines for the application; it can also provide resources on the same rack or a different rack if the specific machine is unavailable. The ApplicationMaster might, at times, due to vagaries of busyness of the cluster, receive resources that are not the most appropriate; it can then reject them by returning them to the Scheduler without being *charged* for them.

A very important improvement over Hadoop Map-Reduce is that the cluster resources are no longer split into map slots and reduce slots. This has a huge impact on cluster utilization since applications are no longer bottlenecked on slots of either type.

Note: A rogue ApplicationMaster could, in any scenario, request more containers than necessary - the Scheduler uses application-limits, user-limits, queue-limits etc. to protect the cluster from abuse.

Pros and Cons

The main advantage of the proposed model is that it is extremely tight in terms of the amount of state necessary, per application, on the ResourceManager for scheduling and the amount of information passed around between the ApplicationMaster & ResourceManager. This is crucial for scaling the ResourceManager. The amount of information, per application, in this model is always O(cluster size), where-as, in the current Hadoop Map-Reduce JobTracker it is O(number of tasks) which could run into hundreds of thousands of tasks.

The main issue with the proposed resource model is that there is a **loss of information** in the *translation* from splits to hosts/racks as described above. This translation is one-way and irrevocable, and the ResourceManager has no concept of *relationships* between the resource asks for e.g. it the above e.g. there is no need for h46 if h43 and h32 were allocated to the application.

To get around the above disadvantage and *future proof* the Scheduler we propose a simple extension - the idea is to add a complementary Scheduler component to the ApplicationMaster, which performs the translation without the help/knowledge of the ApplicationMaster who continues to think in terms of tasks/splits without the translation. This Scheduler component in the ApplicationMaster is paired with the Scheduler and we could, conceivably, remove this component in the future without affecting any of the implementations of the ApplicationMaster.

Scheduling

The Scheduler aggregates resource requests from all the running applications to build a global plan to allocate resources. The Scheduler then allocates resources based on application specific constraints such as appropriate machines and global constraints such as capacities of the application, queue, user etc.

The Scheduler uses familiar notions of Capacity and Capacity Guarantees as the *policy* for arbitrating resources among competing applications.
The scheduling algorithm is straightforward:

- Pick the most under-served queue in the system
- Pick the highest priority job in that queue
- Serve the job's resource asks

Scheduler API

There is a single API between the YARN Scheduler and the ApplicationMaster:

```
Response allocate (List<ResourceRequest> ask, List<Container> release)
```

The AM ask for specific resources via a list of ResourceRequests (ask) and releases unnecessary Containers which were allocated by the Scheduler.

The Response contains a *list of newly allocated Containers*, the statuses of application-specific Containers that completed since the previous interaction between the AM and the RM and indicator to application about available headroom for cluster resources. The AM can use the Container statuses to glean information about completed containers and react to failure etc. The headroom can be used by the AM to tune it’s future requests for e.g. the MR AM can use this information to decide to schedule maps and reduces appropriately to avoid deadlocks such as using up all its headroom for reduces etc.

Resource Monitoring

The Scheduler receives periodic information about the resource usages on allocated resources from the NodeManagers. The Scheduler also makes available status of completed Containers to the appropriate ApplicationMaster.

Application Submission

The workflow of application-submission is straightforward:

- The user (usually from the gateway) submits a job to the ApplicationsManager.
    - The user client first gets a new ApplicationID
    - Packages the application definition, uploads to HDFS in *${user}/.staging/${application_id}*

  - Submits the application to ApplicationsManager
- The ApplicationsManager accepts the application -submission
- The ApplicationsManager negotiates the first slot necessary for the ApplicationMaster from the Scheduler and launches the ApplicationMaster
- The ApplicationsManager is also responsible for providing details of the running ApplicationMaster to the client for monitoring application progress etc.

Lifecycle of the ApplicationMaster

The ApplicationsManager is responsible for managing the lifecycle of the ApplicationMaster of all applications in the system.

As described in the section on Application Submission the ApplicationsManager is responsible for starting the ApplicationMaster.

The ApplicationsManager then monitors the ApplicationMaster, who sends periodic heartbeats, and is responsible for ensuring its aliveness, restarting the ApplicationMaster on failure etc.

ApplicationsManager - Components

As described in the previous sections, the primary responsibility of the ApplicationsManager is to manage the lifecycle of the ApplicationMaster.

In order to affect this, the ApplicationsManager has the following components:
- SchedulerNegotiator – Component responsible for negotiating the container for the AM with the Scheduler
- AMContainerManager – Component responsible for starting and stopping the container of the AM by talking to the appropriate NodeManager
- AMMonitor – Component responsible for managing the aliveness of the AM and responsible for restarting the AM if necessary.

Availability

The ResourceManager stores its state in Zookeeper to ensure that it is *highly available*. It can be quickly restarted by relying on the saved state in Zookeeper. Please refer to the YARN availability section for more details.

NodeManager

The per-machine NodeManager is responsible for launching containers for applications once the Scheduler allocates them to the application. The NodeManager is also responsible for ensuring that the allocated containers do not exceed their allocated resource slice on the machine.

The NodeManager is also responsible for setting up the *environment* of the container for the tasks. This includes binaries, jars etc.

The NodeManager also provides a simple service for managing local storage on the node. Applications can continue to use the local storage even when they do not have an active allocation on the node. For e.g. Map-Reduce applications use this service to store the temporary map-outputs and *shuffle* them to the reduce tasks.

ApplicationMaster

The ApplicationMaster is the per-application master who is responsible for negotiating resources from the Scheduler, running the component tasks in appropriate containers in conjunction with the NodeManagers and monitoring the tasks. It reacts to container failures by requesting alternate resources from the Scheduler.

The ApplicationMaster is responsible for computing the resource requirements of the application e.g. *input-splits* for MR applications, and translating them into the protocol understood by the Scheduler.

The ApplicationMaster is also recovering the application on its own failure. The ApplicationsManager only restarts the AM on failure; it is the responsibility of the AM to recover the application from saved persistent state. Of course, the AM can just run the application from the very beginning.

Running Map-Reduce Applications in YARN

This section describes the various components used to run Hadoop Map-Reduce jobs via YARN.

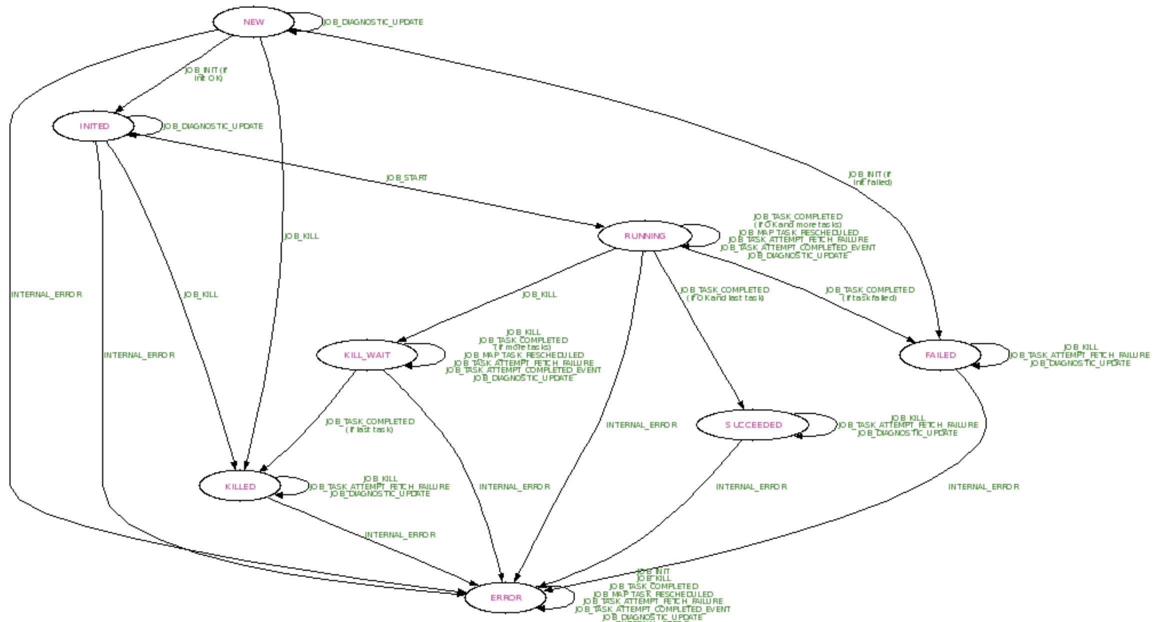
Map-Reduce ApplicationMaster

The Map-Reduce ApplicationMaster is the Map-Reduce specific kind of ApplicationMaster, which is responsible for managing MR jobs.

The primary responsibility of the MR AM is to acquire appropriate resources from the ResourceManager, schedule tasks on allocated containers, monitor the running tasks and manage the lifecycle of the MR job to completion. Another responsibility of the MR AM is to save state of the job to persistent storage so that the MR job can be recovered if the AM itself fails.

The proposal is to use an event-based, Finite State Machine (FSM) in the MR AM to manage the lifecycle and recovery of the MR tasks and job(s). This allows a natural expression of the state of the MR job as a FSM and allows for maintainability and observability.

The following diagrams indicate the FSM for MR job, task and task-attempt:



YARN Job State Machine, 20110328
(app/job/impl/JobImpl.java)

State Transition Diagram for MR Job

YARN Task State Machine, 20110301
(app/job/impl/TaskImpl.java)

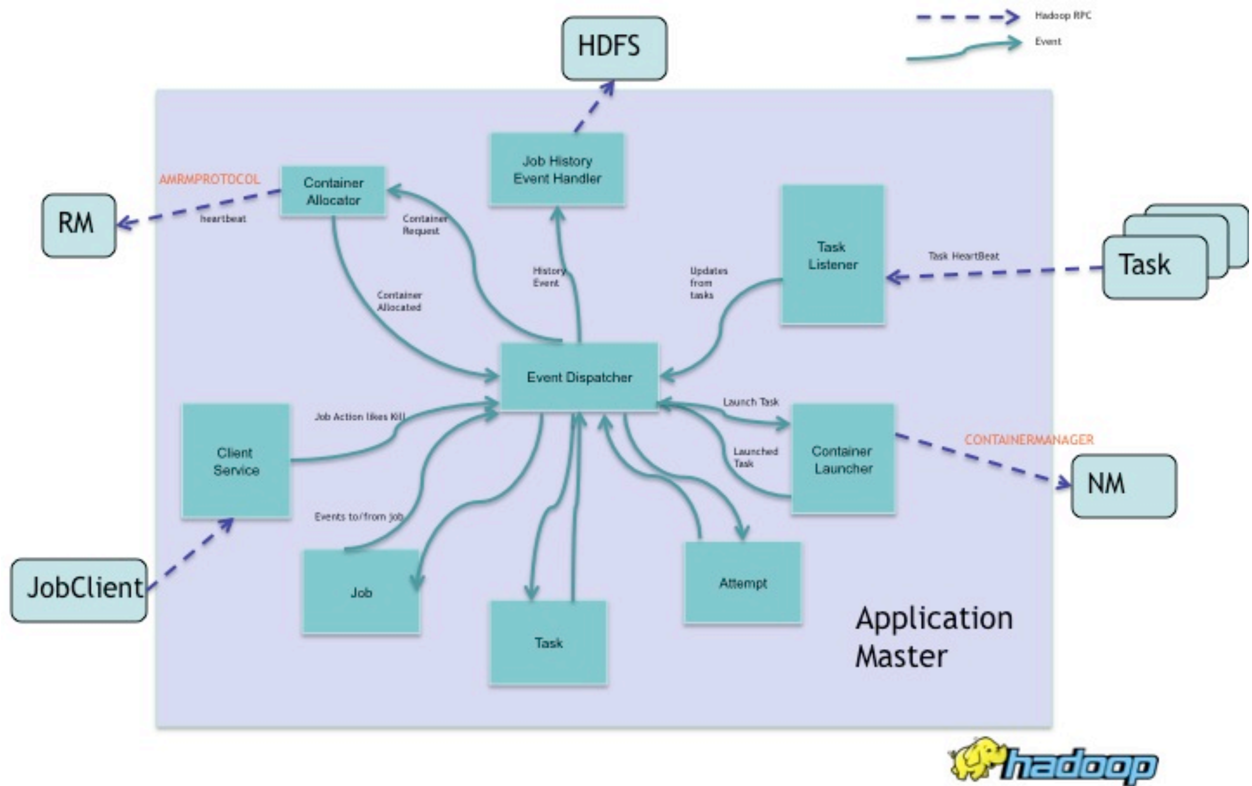
YARN Task Attempt State Machine, 20110323
(app/job/impl/TaskAttemptImpl.java)

The lifecycle of a Map-Reduce job running in YARN looks so:

- Hadoop MR JobClient submits the job to the yarn ResourceManager (ApplicationsManager) rather than to the Hadoop Map-Reduce JobTracker.
- The YARN ASM negotiates the container for the MR AM with the Scheduler and then launches the MR AM for the job.
- The MR AM starts up and registers with the ASM.
- The Hadoop Map-Reduce JobClient polls the ASM to obtain information about the MR AM and then directly talks to the AM for status, counters etc.
- The MR AM computes input-splits and constructs *resource requests* for all maps to the YARN Scheduler.
- The MR AM runs the requisite job setup APIs of the Hadoop MR OutputCommitter for the job.
- The MR AM submits the resource requests for the map/reduce tasks to the YARN Scheduler, gets containers from the RM and schedules appropriate tasks on the obtained containers by working with the NodeManager for each container.
- The MR AM monitors the individual tasks to completion, requests alternate resource if any of the tasks fail or stop responding.
- The MR AM also runs appropriate task cleanup code of completed tasks of the Hadoop MR OutputCommitter.
- Once the entire once map and reduce tasks are complete, the MR AM runs the requisite job commit or abort APIs of the Hadoop MR OutputCommitter for the job.
- The MR AM then exits since the job is complete.

The Map-Reduce ApplicationMaster has the following components:

- Event Dispatcher – The central component that acts as the coordinator that generates events for other components below.
- ContainerAllocator – The component responsible for translating tasks' resource requirements to *resource requests* that can be understood by the YARN Scheduler and negotiates resources with the RM.
- ClientService – The component responsible for responding to the Hadoop Map-Reduce JobClient with job status, counters, progress etc.
- TaskListener – Receives heartbeats from map/reduce tasks.
- TaskUmbilical – The component responsible for receiving heartbeats and status updates form the map and reduce tasks.
- ContainerLauncher – The component responsible for launching containers by working the appropriate NodeManagers.
- JobHistoryEventHandler – Writes job history events to HDFS.
- Job – The component responsible for maintaining the state of the job and component tasks.



Availability

The ApplicationMaster stores its state in HDFS, typically, to ensure that it is *highly available*.

Please refer to the YARN availability section for more details.

YARN Availability

This section describes the design for affecting high availability for YARN.

Failure Scenarios

This section enumerates the various abnormal and failure scenarios for entities in YARN such as ResourceManager, ApplicationMasters, Containers etc. and also highlights the entity responsible for handling the situation.

Scenarios and handling entity:

- Container (task) live, but wedged – AM times out the Container and kills it.
- Container (task) dead – The NM notices that the Container has exited, notifies the RM (Scheduler); the AM picks up the status of the Container during its next interaction with the YARN Scheduler (see section on Scheduler API).
- NM live, but wedged – the RM notices that the NM has timed out, informs all AMs who had live Containers on that node during their next interaction.
- NM dead – the RM notices that the NM has timed out, informs all AMs who had live Containers on that node during their next interaction.
- AM live, but wedged – the ApplicationsManager times out the AM, frees the Container of the RM and restarts the AM by negotiating another Container
- AM dead – the ApplicationsManager times out the AM, frees the Container of the RM and restarts the AM by negotiating another Containers
- RM dead – Please see section on RM restart.

Availability for Map-Reduce Applications and ApplicationMaster

This section describes failover for MR ApplicationMaster and how the MR job is recovered.

As described in the section on the ApplicationsManager (ASM) in the YARN ResourceManager, the ASM is responsible for monitoring and providing high availability for the MR ApplicationMaster or any ApplicationMaster for that matter.

The MR ApplicationMaster is responsible for recovering the specific MR job by itself, the ASM only restarts the MR AM.

We have multiple options for recovering the MR job when the MR AM is restarted:

- Restart the job from scratch.
- Restart only the incomplete map and reduce tasks.
- Point the running map and reduce tasks to the restarted MR AM and run to completion.

Restarting the MR job from scratch is too expensive to do just for the failure of a single container i.e. the one on which the AM is running, and is thus an infeasible option.

Restarting only incomplete map and reduce tasks is attractive since it ensures we do not run the completed tasks, however it still hurts reduces since they might have (nearly) completed the shuffle and are penalized by restarting them.

Changing the running map and reduce tasks to point to the new AM is the most attractive option from the perspective of the job itself, however technically it is a much more challenging task.

Given the above tradeoffs, we are proposing to go with option 2 for YARN-v1.0. It is a simpler implementation, but provides significant benefits. It is very likely that we will implement option 3 for a future version.

The proposal to implement option 2 is to have the MR AM write out a transaction log to HDFS which tracks all of the completed tasks. On restart we can *replay the task completion events* from transaction log, and use the state machine in the MR AM to track the completed tasks. Thus it is fairly straightforward to get the MR AM to run the remaining tasks.

Availability for the YARN ResourceManager

The primary goal here is to be able to quickly restart the ResourceManager and all the running/pending applications in face of catastrophic failures.

Thus **availability** for Yarn has two facets:

- Availability for the ResourceManager.
- Availability for the individual applications and their ApplicationMaster.

The design for high availability for YARN is conceptually very simple - the ResourceManager is responsible for recovering it's own state such as running AMs, allocated resources and NMs. The AM itself is responsible for recovering the state of it's own application. See section on Availability for MR ApplicationMaster for a detailed discussion on recovering MR jobs.

This section describes the design for ensuring that the YARN ResourceManager is highly available.

The YARN ResourceManager has the following state that needs to be available in order to provide high-availability:

- Queue definitions – These are already present in persistent storage such as file-system or LDAP.
- Definition of running and pending applications
- Resource allocations to various applications and Queues.
- Containers already running and allocated to Applications
- Resource allocations on individual NodeManagers.

The proposal is to use ZooKeeper to store state of the YARN ResourceManager that isn't already present in persistent storage i.e. state of AMs and resource allocations to individual Application.

The resource allocations to various NMs and Queues can be quickly rebuilt by scanning allocations to Applications.

ZooKeeper allows for quick failure recovery for the RM and failover via master election.

A simple scheme for this is to create ZooKeeper ephemeral nodes for each of the NodeManagers and one node per allocated container with the following minimal information:

- ApplicationID
- Container capability

An illustrative example:

```
|
+ app_1
|     + <container_for_AM>
|     |
|     + <container_1, host_0, 2G>
|     |
|     + <container_11, host_10, 1G>
|     |
|     + <container_34, host_9, 2G>
|
+ app2
|     + <container_for_AM>
```

```
|      |
|      + <container_5, host_87, 1G>
|      |
|      + <container_67, host_14, 4G>
|      |
|      + <container_87, host_9, 2G>
|      |
```

With this information, the ResourceManager, on a reboot, can quickly rebuild state via ZooKeeper. ZooKeeper is the source of truth for all assignments.

Here is the flow of container assignment.

- ResourceManager gets a request for a container from ApplicationMaster
- ResourceManager creates a znode in ZooKeeper under /app$i with the containerid, node id and resource capability information.
- ResourceManager responds to the ApplicationMaster request of the resource capability.

On release of the containers, these are the steps followed:

- ApplicationMaster sends a `stopContainer()` request to the NodeManager to stop the container, on a failure it retries.
- ApplicationMaster then sends a request to the ResourceManager with a `deallocateContainer(containerid)`
- ResourceManager then removes the container znode from ZooKeeper.

To be able to keep all the container allocation in sync between the ApplicationMaster, ResourceManager and NodeManager, we need a few APIs:

ApplicationMaster uses the `getAllocatedContainers(appid)` to keep itself in sync with the ResourceManager on the set of containers allocated to the ApplicationMaster.

NodeManager keeps itself in sync with heartbeat to the ResourceManager which responds back with containers that the NodeManager should cleanup and remove.

On restart the ResourceManager goes through all the application definitions in HDFS/ZK i.e. /systemdir and assumes all the RUNNING ApplicationMasters are alive. The ApplicationsManager is responsible for restarting any of the failed AMs that do not update the ApplicationsManager as described previously.

The resource manager on startup publishes its host and port on ZooKeeper, under ${yarn-root}/yarn/rm.

```
|
+ yarn
|      |
|      + <rm>
```

This ZooKeeper node is an ephemeral node that gets deleted automatically if the ResourceManager dies. A **backup** ResourceManager can take over in that case via notification from ZooKeeper. Since the ResourceManager state is in ZooKeeper the backup ResourceManager can startup.

All the NodeManagers watch ${yarn-root}/yarn/rm to get notified in case the znode gets deleted or updated.

The ApplicationMaster also watches for the notifications on ${yarn-root}/yarn/rm.

On deletion of the znode NodeManagers and ApplicationMasters get notified of the change. This change in ResourceManager can be notified via ZooKeeper to the yarn components.

YARN Security

This section describes security aspects of YARN.

A hard requirement is for YARN to at least as secure as the current Hadoop Map-Reduce framework.

This implies several aspects such as:

- Kerberos based strong authentication across the board.
- YARN daemons such as RM and NM should run as secure, non-root Unix users.
- Individual applications run as the actual user who submitted the application and they have secure access to data on HDFS etc.
- Support for super-users for frameworks such as Oozie.
- Comparable authorization for queues, admin tools etc.

The proposal is to use the existing security mechanisms and components from Hadoop to effect security for YARN.

The only additional requirement for YARN is that NodeManagers can securely authenticate incoming requests to allocate and start containers by ensuring that the Containers were actually allocated by the ResourceManager.

The proposal is to use a shared secret key between the RM and the NM. The RM uses the key to sign an authorized ContainerToken with the ContainerID, ApplicationID and allocated resource capability with the key. The NM can use the shared key to decode the ContainerToken and verify the request.

Security for ZooKeeper

ZooKeeper has pluggable security that can use either YCA or shared secret authentication. It has ACLs for authorization.

The RM node in ZooKeeper for discovering the RM and the application container assignment mentioned in section on YARN RM Availability are writable by the ResourceManager and only readable by others. YCA is used as an authentication system.

Note that since ZooKeeper doesn't yet support Kerberos, the individual ApplicationMasters and tasks are not allowed to write to ZooKeeper, all writes are done by the RM itself.