"YARN Federation" *YARN-2915*

Design Document

Design Contributors

Mitch Arene

Kishore Chaliparambil

Carlo Curino

Chris Douglas

Giovanni Matteo Fumarola

Solom Heddaya

Subru Krishnan

Raghu Ramakrishnan

Sriram Rao

Sarvesh Sakalanaga

Radhey Shah

Bin Shi

Brook Zhou

Contents

1 Introduction

YARN is known to scale to thousands of nodes. The scalability of YARN is determined by the Resource Manager, and is proportional to number of nodes, number of active applications, and frequency of heartbeat (of both nodes and applications). Lowering heartbeat can provide scalability increase, but is detrimental to utilization (see old Hadoop 1.x experience).

This document described a federation-based approach to scale a single YARN cluster to tens of thousands of nodes, by federating multiple YARN sub-clusters. The proposed approach is to divide a large (10-100k) nodes cluster into smaller units called sub-clusters, each with its own YARN RM and compute nodes. The federation system will stitch these sub-clusters together and make them appear as one large YARN cluster to the applications.

The applications running in this federated environment will see a single massive YARN cluster and will be able to schedule tasks on any node of the federated cluster. Under the hood, the federation system will negotiate with sub-clusters resource managers and provide resources to the application. The goal is to allow an individual job to "span" sub-clusters seamlessly.

This design is structurally scalable, as we bound the number of nodes each RM is responsible for, and appropriate policies, will try to ensure that the majority of applications will reside within a single sub-cluster, thus the number of applications each RM will see is also bounded. This means we could almost linearly scale, by simply adding sub-clusters (as very little coordination is needed across them).

This architecture can provide very tight enforcement of scheduling invariants within each sub-cluster (simply inherits from YARN), while continuous rebalancing across subcluster will enforce (less strictly) that these properties are also respected at a global level (e.g., if a sub-cluster loses a large number of nodes, we could re-map queues to other sub-clusters to ensure users running on the impaired sub-cluster are not unfairly affected).

Federation is being design as a "layer" atop of existing YARN codebase, with limited changes in the core YARN mechanisms.

Assumptions

- We assume reasonably good connectivity across sub-clusters (e.g., we are not looking to federate across DC yet, though future investigations of this are not excluded).
- We rely on HDFS federation (or equivalently scalable DFS solutions) to take care of scalability of the store side.

2 YARN Federation Architecture

In this section we will discuss high level architecture and main components involved in federation.

OSS YARN has been known to scale up to about few thousand nodes. The proposed architecture leverages the notion of *federating* a number of such smaller YARN clusters, referred to as sub-clusters, into a larger federated YARN cluster comprising of tens of thousands of nodes.

The applications running in this federated environment see a unified large YARN cluster and will be able to schedule tasks on any nodes cluster. Under the hood, the federation system will negotiate with sub-clusters RMs and provide resources to the application. The logical architecture in Figure 1 shows the main components that comprise the federated cluster, which are described below.

2.1 *Logical Architecture*

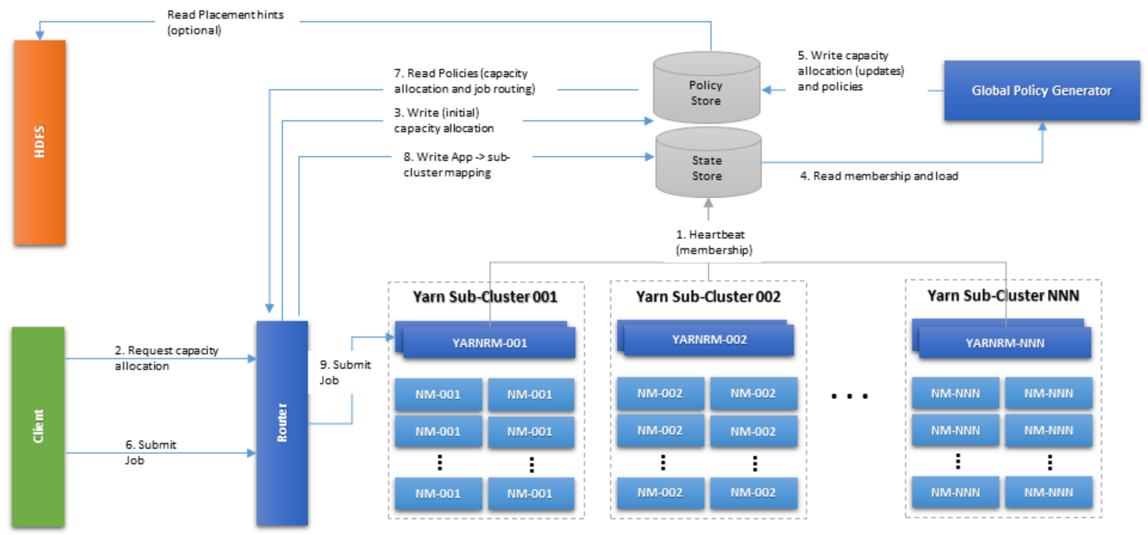
Read Placement hints (optional)
HDFS
7. Read Policies (capacity allocation and job routing)
3. Write (initial) capacity allocation
8. Write App -> sub-cluster mapping
Policy Store
State Store
5. Write capacity allocation (updates) and policies
Global Policy Generator
4. Read membership and load
1. Heartbeat (membership)
Yarn Sub-Cluster 001
Yarn Sub-Cluster 002
Yarn Sub-Cluster NNN
YARNRM-001
YARNRM-002
YARNRM-NNN
NM-001
NM-002
NM-NNN
Client
2. Request capacity allocation
6. Submit Job
Router
9. Submit Job

Figure 1 Logical Architecture

2.1.1 YARN Sub-Cluster

A sub-cluster is a YARN cluster with up to few thousands nodes. The exact size of the sub-cluster will be determined considering ease of deployment/maintenance, alignment with network or availability zones and general best practices.

The sub-cluster YARN RM will run with work-preserving *high-availability* turned-on, i.e., we should be able to tolerate YARN RM, NM failures with minimal disruption. If the entire sub-cluster is compromised, external mechanisms will ensure that jobs are resubmitted in a separate sub-clusters (this could eventually be included in the federation design).

Sub-cluster is also the *scalability unit* in a federated environment. We can scale out the federated environment by adding one or more sub-clusters. Furthermore separate “independent” clusters can decide to participate to the federation by contributing a fraction of their capacity, i.e. each sub-cluster remains a fully functional YARN cluster, which is also contributing to the federation part of all of its capacity.

Note: by design each sub-cluster is a fully functional YARN RM, and its contribution to the federation can be set to be only a fraction of its overall capacity, i.e. a sub-cluster can have a “partial” commitment to the federation, while retaining the ability to give out part of its capacity in a completely local way.

2.1.2 Router (*YARN-3659*)

YARN applications are submitted to one of the Routers, which in turn applies a routing policy (obtained from the Policy Store), queries the State Store for the sub-cluster URL and redirects the application submission request to the appropriate sub-cluster RM. We call the sub-cluster where the job is started the *"home sub-cluster"*, and we call *"secondary sub-clusters"* all other sub-cluster a job is spanning on.

The Router exposes the ApplicationClientProtocol to the outside world, transparently hiding the presence of multiple RMs. To achieve this the Router also persists the mapping between the application and its home sub-cluster into the State Store. This allows Routers to be soft-state while supporting user requests cheaply, as any Router can recover this application to home sub-cluster mapping and direct requests to the right RM without broadcasting them. For performance caching and session stickiness might be advisable.

At any one time, a job can span across one home sub-cluster and multiple secondary sub-clusters, but the policies we are devising will try to limit the footprint of each job to minimize overhead on the scheduling infrastructure (more in section on scalability/load).

2.1.3 AMRMProxy (YARN-3666)

The AMRMProxy is a key component to allow the application to scale and run across sub-clusters. The AMRMProxy runs on all the NM machines and acts as a proxy to the YARN RM for the AMs by implementing the ApplicationMasterProtocol. Applications will not be allowed to communicate with the sub-cluster RMs directly. They are forced by the system to connect only to the AMRMProxy endpoint, which would provide transparent access to multiple YARN RMs (by dynamically routing/splitting/merging the communications).

2.1.4 Global Policy Generator (GPG) (*YARN-3660*)

Global Policy Generator overlooks the entire federation and ensures that the system is configured and tuned properly all the time.

A key design point is that the cluster availability does not depends on an always-on GPG. The GPG operates continuously but out-of-band from all cluster operations, and provide us with a unique vantage point, that allows to enforce global invariants, affect load balancing, trigger draining of sub-clusters that will undergo maintenance, etc.

More precisely the GPG will update user capacity allocation-to-subcluster mappings, and more rarely change the policies run in Routers, AMRMProxy (and possible RMs).

In case the GPG is not-available, cluster operations will continue as of the last time the GPG published policies, and while a long-term unavailability might mean some of the desirable properties of balance, optimal cluster utilization and global invariants might drift away, compute and access to data will not be compromised.

NOTE: In the current prototype the GPG is a manual tuning process, simply exposed via a CLI (YARN-3657).

2.1.5 Federation State Store (YARN-3662, YARN-3663)

The Federation State defines the additional state that needs to be maintained to loosely couple multiple individual sub-clusters into a single large federated cluster. This includes the following information:

2.1.6 Sub-cluster Membership (YARN-3665)

The member YARN RMs continuously heartbeat to the state store to keep alive and publish their current capability/load information. This information is used by the Global Policy Generator (GPG) to make proper policy decisions. Also this information can be used by routers to select the best home sub-cluster. This mechanism allows us to dynamically grow/shrink the “cluster fleet” by adding or removing sub-clusters. This also allows for easy maintenance of each sub-cluster. This is new functionality that needs to be added to the YARN RM but the mechanisms are well understood as it’s similar to individual YARN RM HA.

2.1.7 Application’s Home Sub-cluster

The sub-cluster on which the Application Master (AM) runs is called the Application’s “home sub-cluster”. The AM is not limited to resources from the home sub-cluster but can also request resources from other sub-clusters, referred to as secondary sub-clusters.

The federated environment will be configured and tuned periodically such that when an AM is placed on a sub-cluster, it should be able to find most of the resources on the home sub-cluster. Only in certain cases it should need to ask for resources from other sub-clusters.

Recent analysis of failure modes suggest that we should also maintain an explicit mapping between the notion of an “external App id” and the “internal App id”. This would allow us to hide some class of local failures (e.g., one RM is not reachable and we need to resubmit with a new app id).

2.1.8 Federation Policy Store (YARN-3664, YARN-3663)

The federation Policy Store is a logically separate store (while it might be backed by the same physical component), which contains information about the capacity allocations made by users, their mapping to sub-clusters and the policies that each of the components (Router, AMRMPRoxy, RMs) should enforce.

2.1.9 Capacity Allocation across sub-clusters (YARN-3658)

An important problem is how to map federation-level capacity allocations to sub-clusters level allocations (our enforcement mechanism). The two existing options are:

1) Leveraging the queue mechanism and map federation-level queues to sub-cluster level queues.
2) Leveraging reservations (YARN-1051) and map those to sub-clusters.

Reservations have the advantage of being (by construction) more dynamic, the reason this is relevant is that capacity in various sub-cluster will come and go due to failures, and to enforce global invariants, we will need to grow/shrink the capacity allocated to users/groups in each subcluster to guarantee global invariants. Reservations are expressed in absolute terms (e.g., 100 containers, vs 0.2% of my cluster), and are already equipped with mechanisms to very dynamically add/remove/resize reservations. As such they are likely a good building block.

Crisp support of the semantics of hierarchies of queues/reservation across sub-clusters is an open problem.

NOTE: While most of the other mechanisms discussed in this document have been already prototyped, this is subject of ongoing investigation.

2.2 Running Applications (across Sub-Clusters)

When an application is submitted, the system will determine the most appropriate sub-cluster to run the application, which we call as the application's *home sub-cluster*. All the communications from the AM to the RM will be proxied via the AMRMProxy running locally on the AM machine. AMRMProxy exposes the same *ApplicationMasterService* protocol endpoint as the YARN RM. The AM can request containers using the locality information exposed by the storage layer. In ideal case, the application will be placed on a sub-cluster where all the resources and data required by the application will be available, but if it does need containers on nodes in other sub-clusters, AMRMProxy will negotiate with the RMs of those sub-clusters transparently and provide the resources to the application, thereby enabling the application to view the entire federated environment as one massive YARN cluster. AMRMProxy, Global Policy Generator (GPG) and Router work together to make this happen seamlessly.

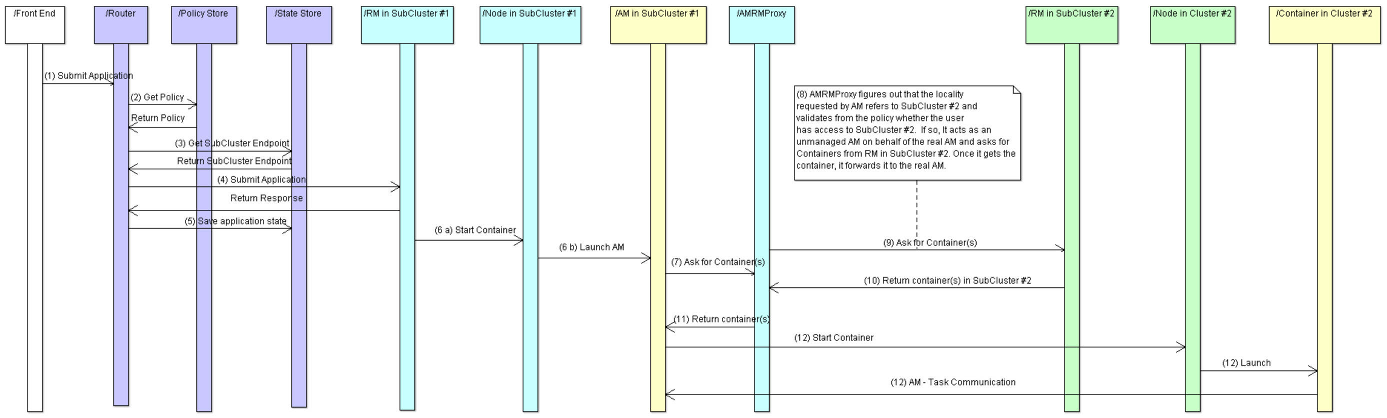
/Front End
/Router
/Policy Store
/State Store
/RM in SubCluster #1
/Node in SubCluster #1
/AM in SubCluster #1
/AMRMProxy
/RM in SubCluster #2
/Node in Cluster #2
/Container in Cluster #2
(1) Submit Application
(2) Get Policy
Return Policy
(3) Get SubCluster Endpoint
Return SubCluster Endpoint
(4) Submit Application
Return Response
(5) Save application state
(6 a) Start Container
(6 b) Launch AM
(7) Ask for Container(s)
(8) AMRMProxy figures out that the locality requested by AM refers to SubCluster #2 and validates from the policy whether the user has access to SubCluster #2. If so, It acts as an unmanaged AM on behalf of the real AM and asks for Containers from RM in SubCluster #2. Once it gets the container, it forwards it to the real AM.
(9) Ask for Container(s)
(10) Return container(s) in SubCluster #2
(11) Return container(s)
(12) Start Container
(12) Launch
(12) AM - Task Communication

Figure 2 Job execution flow

Figure 2 Job execution flow shows the interactions between components during job execution

1. The Router receives an application submission request that is complaint to the YARN Application Client Protocol.
2. The router interrogates a routing table / policy to choose the "home RM" for the job.
3. The router queries the membership state to determine the endpoint of the *home RM*.
4. The router then redirects the application submission request to the *home RM*.
5. The router updates the application state with the *home RM identifier*.
6. Once the application is submitted to the *home RM*, the stock YARN flow is triggered, i.e. the application is added to the scheduler queue and its AM started in the home sub-cluster, on the first NodeManager that has available resources.
    a. During this process, the AM environment is modified by indicating that the address of the AMRMProxy as the YARN RM to talk to.
    b. The security tokens are also modified, so that the AM can only talk with the AMRMProxy. Any future communication from AM to the YARN RM is mediated by the AMRMProxy.
7. The AM will then request containers using the locality information exposed by HDFS.
8. Based on a policy the AMRMProxy can impersonate the AM on other sub-clusters, by submitting an Unmanaged AM, and by forwarding the AM heartbeats to both clusters.
9. The AMRMProxy will use both locality information and a pluggable policy pushed down by the Global Policy Generator to decide whether to forward the resource requests received by the AM to the Home

RM or to one (or more) Secondary RMs. In Figure 2 Job execution flow, we show the case in which the AMRMProxy decides to forward the request to the secondary RM.

10. The secondary RM will provide the AMRMProxy with valid container tokens to start a new container on some node in its sub-cluster (#2 in Figure 3). This mechanism ensures that each sub-cluster uses its own security tokens and avoids the need for a cluster wide shared secret to create tokens.
11. The AMRMProxy forwards the allocation response back to the AM.
12. The AM starts the container on the target *NodeManager* (on sub-cluster 2) using the standard YARN protocols.

2.2.1 Role of AMRMProxy

1. Protect the sub-cluster YARN RMs from misbehaving AMs. The AMRMProxy can prevent DDOS attacks by throttling/killing AMs that are asking too many resources.
2. Mask the multiple YARN RMs in the cluster, and can transparently allow the AM to span across sub-clusters. All container allocations is done by the YARN *RM framework* that consists of the AMRMProxy fronting the home and other sub-cluster RMs.
3. Intercepts all the requests, thus it can enforce application quotas, which would not be enforceable by sub-cluster RM (as each only see a fraction of the AM requests).
4. The AMRMProxy can enforce load-balancing / overflow policies.

2.2.2 Role of Global Policy Generator (GPG)

1. Overlooks the entire federation and tunes the system.
2. Creates reservations (dynamic queues) on the sub-clusters for each account.
3. Creates routing policies for the Router service.
4. Creates application scheduling policies for the AMRMProxy service running on every node.

3 Detailed Design

So far we discussed the structural components and their responsibility at a high level. In this section we will discuss design and implementation of components and services in more details.

3.1 Router

The router is new stateless YARN component which is the entry point to the federated cluster. It can be deployed on multiple nodes behind a Virtual IP (VIP). The router implements the entire ApplicationClientProtocol of YARN, which allows users to request and update reservations, submit and kill applications, and request status on running applications. This is done by appropriately decomposing user initiated requests in one or more requests that are forwarded to a subset of the RMs, and combining the results.

The router is responsible for:

1) Generating globally unique application identifiers.
2) Determining the home sub-cluster by applying an AMRoutingPolicy.
3) Routing the submitApplication request to the home-RM and storing in the State Store the mapping between app and home-RM (the home-RM URL is obtained from the StateStore membership table).
4) Routing subsequent requests regarding this application requests to the right RM.
5) As the job might be consuming resources across sub-clusters (4) above might require broadcasting to multiple RMs and combining answers in order to maintain transparency.

3.1.1 AM Routing

Figure 3 illustrates the flow of how the router utilizes the policy to determine the sub-cluster to redirect the user application submission request.

1. The Router queries the PolicyStore to retrieve the AMRoutingPolicy and reservation to sub-cluster mapping for the user account who initiated the application submission. Optionally router can retrieve all the policies and cache it for performance optimization.
2. The PolicyStore returns the assigned policy to the router. This is in compact serialized format, in our case using *protocol buffer*. So this is deserialized into a *FederationPolicyInstance* object before returning to the router.
3. The router passes the above returned object to its configured interpreter which determines the next sub-cluster ID.
4. The sub-cluster ID is returned to the router.
5. The router forwards the application submission request to the sub-cluster selected previously (by retrieving the corresponding sub-cluster endpoint from state store or its internal cache).
6. The router forwards any response from the sub-cluster RM back to the client.

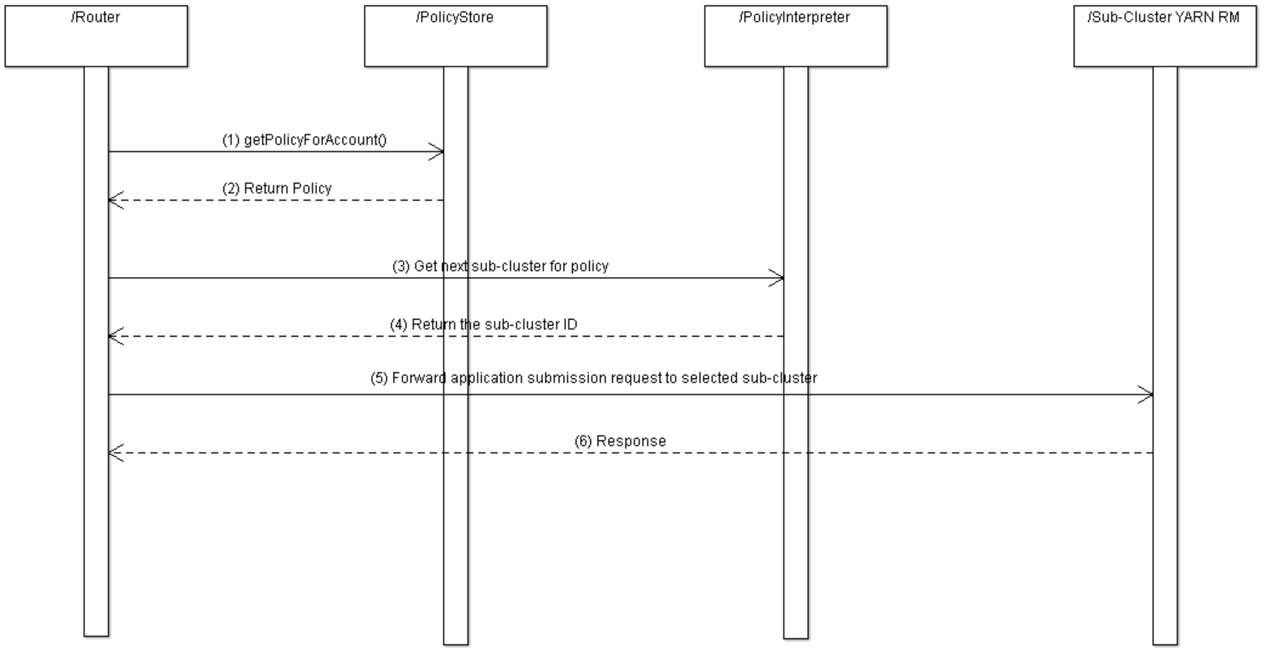
/Router
/PolicyStore
/PolicyInterpreter
/Sub-Cluster YARN RM
(1) getPolicyForAccount()
(2) Return Policy
(3) Get next sub-cluster for policy
(4) Return the sub-cluster ID
(5) Forward application submission request to selected sub-cluster
(6) Response

Figure 3 Router sequence diagram

Note: while the diagram in Fig. 3 shows several roundtrips to State Store and Policy Store, many of those interactions can be easily cached. For example, the mapping application-subcluster can be easily kept in the main-memory of the router. Upon router restart, or if we talk with a router that never saw requests for this app, the request can be sent to the sub-cluster. If the information ever becomes stale it is easy to detect (the RM responds by saying the application is not in this subcluster), and then we can force a refresh by asking the State Store.

Note: The AMRMProxy will follow the same sequence but will additional apply the locality constraints (if any) specified by the AM for forwarding the resource allocation requests. We expect our understanding of the policies space to evolve drastically as we start operating clusters and learn operational requirements and priorities.

Example of AMRoutingPolicy

We are considering possibly different AMRoutingPolicy(s), here example of few such policies:

- ***Simple Priority***: The most basic policy interpreter will be based on a priority order determined by the user reservation on the sub-clusters. Assume a user has reservation on sub-clusters 1, 3, 4. The policy will try to start the AM on sub-cluster 1 and fallback on 3 and 4 if sub-cluster 1 is not available.
- ***Simple Weighting***: Another simple policy interpreter can allows us to spread the job among the sub-clusters the user has rights to run on in a way that is weighted by the size of reservation on each cluster. E.g., if the user has 70 machines reserved on sub-cluster 1, 20 machines on sub-cluster 3 and 10 machines on sub-cluster 4, jobs will be started with a 70% chance on sub-cluster 1, etc.
- ***Load-balancing***: This option is similar to the Simple Weighting but it also leverages information on current resource utilization on the sub-cluster for this user. I.e., if I am using 68 out of the 70 machines on sub-cluster one, and 5/10 on sub-cluster 4, I will be more likely routed to 4 than 1.

3.2 AMRMProxy

The local Resource Manager proxy (also called as AMRMProxy) is a service running on each node (or once per rack/pod) and hosted as a YARN NM auxiliary service. AMRMProxy exposes an endpoint that listens for ApplicationMasterService protocol messages similar to the YARN RM. All AMs running on a node will be configured to connect to this AMRMProxy endpoint. AMRMProxy will thus act as a proxy between AMs and the YARN RM.

3.2.1 ResourceRequestRoutingPolicy

AMRMProxy will use the following table to decide when to send the request to the Home RM and when to negotiate resources from other sub-cluster resource managers.

Is node specified in request?	**Is node on home cluster?**	**Relax locality?**	**AMRMProxy Policy**	**Container location**
Yes	Yes	No	Check load and throttling and proxy request to home RM	Home sub-cluster
Yes	No	No	Check if AM has access to the requested sub-cluster. If yes, act as an un-managed AM and proxy request to secondary sub-cluster	Secondary sub-cluster
Yes	Yes/No	Yes	Check load and throttling and proxy request to home/secondary RM	Home sub-cluster

Is node specified in request?	Is node on home cluster?	Relax locality?	AMRMProxy Policy	Container location
			based on headroom and access rights	
No	Yes/No	Yes/No	Check load and throttling and proxy request to home RM if possible. If not, act as an un-managed AM and proxy request to one of the secondary RM(s).	Home or secondary sub-cluster

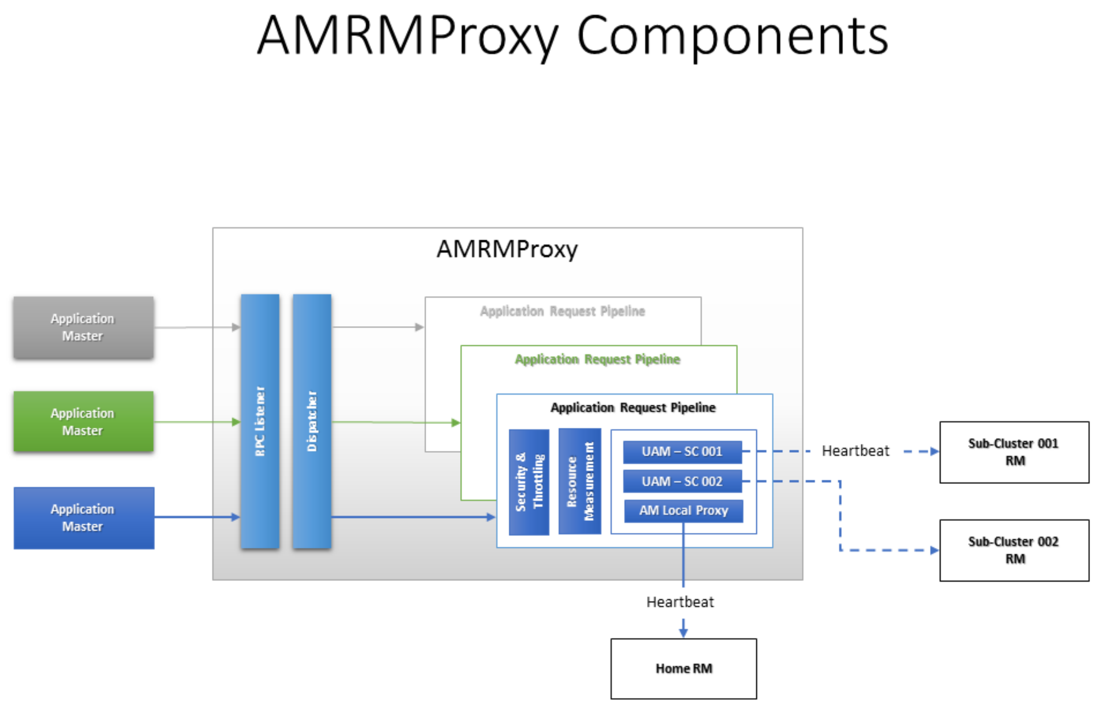

Figure 4 AMRMProxy execution pipeline

AMs always connect to the AMRMProxy end point for registering the application and sending heartbeats. The AMRMProxy will create an application request processing pipeline for each AM as shown in Figure 4. This allows the AMRMProxy to keep information about each AM in a separate pipeline instance. The pipeline also provides an easy way to plug-in request interceptors to add custom logic. For example in future, we will need modules to throttle request, modules for resource measurement, auditing, etc. The last interceptor in the pipeline is responsible for connecting to the real YARN resource managers. Since an AM can request containers on one or more sub-clusters, the last interceptor will have one or more Unmanaged AM instances, one for each sub-cluster in addition to an AM local proxy to connect to the Home Resource Manager.

3.3 State Store

By design, our federation relies on a centralized component for persistency and coordination of key information. We call this component the State Store. The design principle we are following is that we will only maintain minimum additional state needed to orchestrate YARN federation. For state which is currently maintained by the YARN Resource Manager (RM), we will reuse the information by broadcasting queries and returning the aggregated and/or filtered response. The intent is to minimize maintenance and update costs of the federation state store, ensure single source of truth for all state and prevent divergence from stock YARN. In this document, we describe the high level APIs to access the state store.

3.3.1 State Store APIs

The Federation State encapsulates all the state that is required to federate multiple individual YARN sub-clusters into a unified large cluster for the end user. This is composed of the following three logical sub-states:

FederationMembershipState

It maintains the membership information of all subcluster(s) that are participating in federation. This includes the capabilities of each subcluster as they can be heterogeneous. Sub-clusters are free to join and leave independently simply by notifying the FederationMembershipState. The APIs exposed to achieve the same are:

- registerRouter (): Routers on startup request an identifier from the membership state store. This will be unique across multiple instances of the router in the federated cluster. The identifier is valid for the lifetime of the router and is regenerated on every (re)start of the router. The router(s) use this identifier as a seed to generate globally unique application ids for the user.
- registerSubCluster (): Subcluster RM(s) use this API to register its participation in federation by publishing their capabilities and scheduler URL. This is typically done during initialization or restart/failover of the RM. Upon successful registration, an identifier for the subcluster which is unique across the federated cluster is returned. The identifier is static, i.e. preserved across restarts and failover.
- deregisterSubCluster (): Subcluster RM(s) use this API to withdraw participation from federation. This is typically done during shutdown of the subcluster for maintenance or elastic scale down of the federated cluster.
- subClusterHeartbeat ():API for periodic heartbeat from a subcluster RM participating in federation to indicate liveliness. The heartbeat publishes the current capabilities as the cluster state is dynamic due to load and change in number of nodes based on scheduled/unscheduled maintenance. The response of the heartbeat can be used to pushdown policies in future.
- getSubClusterInfo (): This is the API that will be used by consumers of membership state like the router to obtain the information about a subcluster that's taking part in federation. The information includes the subcluster endpoint and current capabilities which in conjunction with the policies can be used to make scheduling decisions.

FederationApplicationState

It maintains the information of all applications that have been submitted to the federated cluster. The APIs defined for applications are:

- registerApplication (): The API used by the router to register a "home" subcluster for a newly submitted application.
- achiveApplication (): The API to update an application in the state store. This will be used post completion (success/failure) of the application and in combination with archival policy.

- getHomeSubClusterForApp () : This API returns the "home" subcluster for a previously submitted application. The router uses the returned subcluster identifier to address user queries to the right subcluster.

3.4 Policy Store

The Federation system by design is the composition of loosely coupled independent sub-clusters into a single large cluster. The composition is driven by policies to ensure that the system is configured and tuned optimally over time. The policy store captures "policies", i.e., expected behavior of subsystem components. We maintain policies for three subsystems:

- Router for determining the home sub-cluster of a newly submitted app, and placing reservation in sub-clusters
- AMRMProxy for ascertaining the sub-cluster RM to redirect the resource allocation requests from an application's AM.
- ResourceManager: to enforce reservation in each sub-cluster

Abstractly each policy can be represented by a choice of **interpreter** (i.e., an algorithm) and **a parameterization** of said algorithm. We already discussed policies for Routers and AMRMProxy in section 4.2 and 4.3. For RMs the choice of interpreter consists on picking a specific ReservationSubsystem which fully determines the algorithmic behavior, and the parameterization is an RDL expression capturing the reservation to be enforced locally by this RM.

In the remainder, we describe high level representation for our policies.

3.4.1 Policy Store APIs

The APIs exposed by the store are:

- getPolicyForAccount (): The router/AMRMProxy/RM use this API to retrieve the interpreter and parametrization they are supposed to enforce.
- getAllPolicies (): The router/AMRMProxy/RM use this API to retrieve all policies (if convenient for caching).
- updatePolicies(): the API for the GPG to update policies

3.5 Capacity allocation across sub-clusters

We now present an initial idea of how to support capacity allocation (and rebalancing) across sub-clusters, by leveraging the existing "reservation" APIs for YARN.

3.5.1 *Scenario*

How do we ensure global constraints across the federated sub-clusters? For e.g.: A user asks for:

"100 containers with 2 cores, 4GB of RAM each, and 100GB of (compute tier) storage".

Can we commit to deliver this amount of resources (i.e., shall we accept this request)? Where in the federation do we have the machines to satisfy this request? What happens in case of failures?

We start by defining the language of reservation we can accept, then propose an architecture to handle the requests, and then briefly explore the policy space to map tenants to sub-clusters.

3.5.2 Reservation APIs

In the fullness of time, we aim at supporting the entire reservation APIs available in YARN 2.6 as part of the ApplicationClientProtocol. To start we will consider a subset of the language with limited flexibility, but we want the architecture to enable smooth evolution towards the complete language (and future versions of it).

Initial Language Expressivity The initial language we support is *without* any flexibility, i.e., the user can ask for an exact number of concurrent containers, with certain amount to resources, for a chosen period of time. E.g., 100 containers with 2 cores, 4GB of RAM each. This is akin to a normal YARN queue. Reservations can be shared by multiple jobs, like a queue, or can be obtain on a per-job basis to provide completely dedicated resources to an important production job. As YARN provide dynamic container allocation among jobs sharing a reservation, per-job reservations can be used to guarantee more consistent runs for production jobs.

Next step is to allow users to express time-ranges for this reservations. This is already supported by the existing YARN reservation subsystem, and it's aligned with the federation architecture. Further down the road is enabling the full expressivity of reservation APIs. The architecture discussed in the next section, will allow this, but we are not likely to enable this from the start.

3.5.3 How to support reservations in a fault-tolerant way

In order to provide reservations in a federated environment we need to solve the problem of mapping reservation to sub-clusters.

With Reference to Figure 5 we propose the following design.

1) The Router receives from the user a request for a reservation (standard YARN ApplicationSubmissionProtocol.submitReservation(…))
2) The Router queries the Policy Store and obtains a current copy of the plans for each sub-cluster (this contain a summarized version of committed resources over-time). To support the initial simplified language, plans don't change over-time and can be represented with only few integers.
3) The Router runs an *AdmissionPolicy*, akin to YARN's reservation algorithms against the overall capacity available across cluster, and determines whether an allocation is possible (and in case of a richer language, in when in the plan the reservation should be allocated)
4) The Router runs a *PlacementPolicy* that maps the acceptable reservation onto the sub-clusters.
5) The Router commits the above by writing to the Policy Store the following items: the reservation acceptance, its placement on sub-clusters, and updates the "plans" for each RM
6) Upon success, the router returns to the user a ReservationID that the user can use when submitting jobs to redeem protected access to the reservation resources.
7) On heartbeats the RMs of each sub-cluster learn their role in supporting the reservation and publish it to the local plan (again borrowed as-is from YARN 2.6 reservation mechanism)

The transactional nature of the accesses allow multiple routers to be concurrently receiving reservation requests and do planning. The GPG operates out of band and can edit these decisions based on global cluster conditions, capacity impairment or other more sophisticated algorithms.

This design tolerates several faults naturally. The GPG is operating out of band, and is not required for basic operations. Each Router can fail and if at least one Router is operational we can accept reservations. The State and Policy Store can be temporarily unavailable, but since the RMs have downloaded a copy of their plan, they can continue operate and enforce that, while the Store recover. No new reservation can be accepted while the Policy Store is not available (we could further ameliorate the system availability in this regard, likely at the cost of consistency).

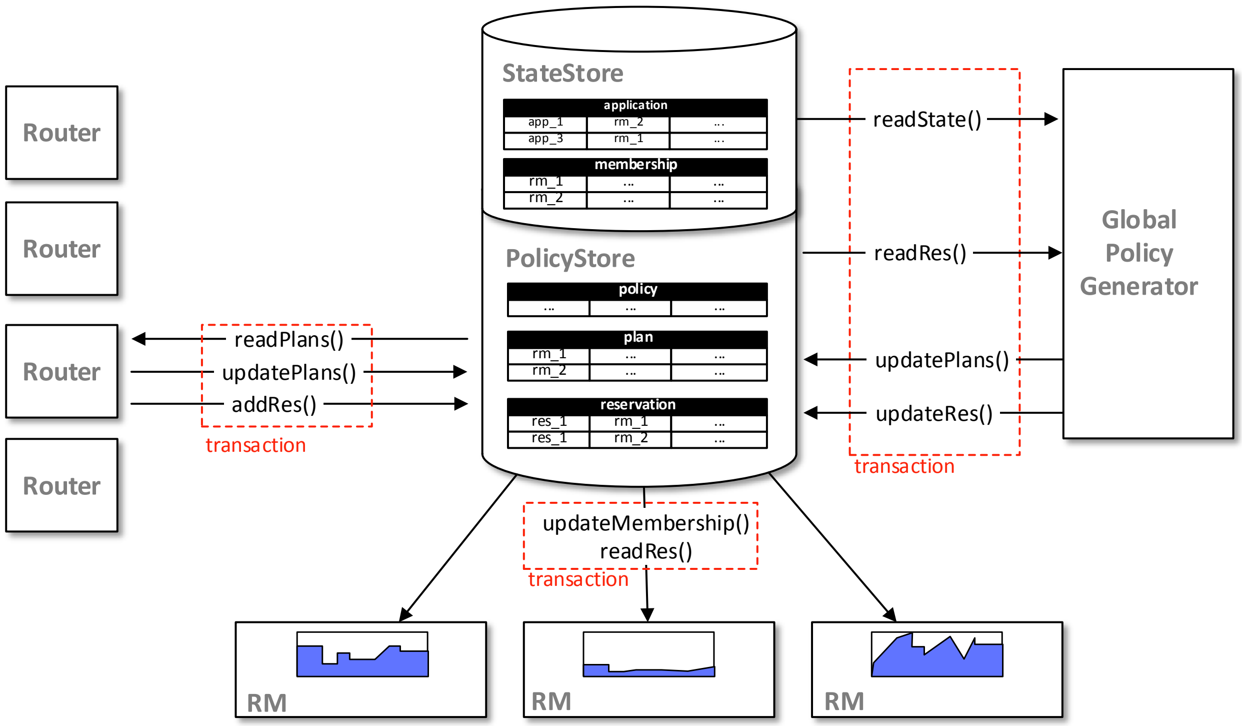

Figure 5 How to support reservation in a federated environment

3.5.4 Current Limitations

3.5.4.1 Support of Hierarchies

The reservation based approach leverages existing building blocks in YARN, and enable us to naturally expand to richer language of reservation in the future. The current limitation is that YARN does *not* support hierarchical reservations. Extending the reservation mechanism in YARN to support hierarchies is rather straightforward for the class of reservation we are considering. What remains unclear is how hard it is to support proper hierarchies across sub-clusters (especially for hierarchies which are mapped in non-uniform ways across clusters). This is independent of the enforcement mechanism (queues or reservations).