[YARN] First-class and simplified support for services

Vinod Kumar Vavilapalli with Sidharta Seethana, Gour Saha, Varun Vasudev, Billie Rinaldi and Jon Maron

Table of Contents

(1) Introduction

An existing umbrella ticket, [YARN-896] [Roll up for long-lived services in YARN], focused on getting the ball rolling on the support for services (long running applications) on YARN. Standing on top of functionality already built for YARN or running regular / short-running apps, it identified an important feature-set needed to enable services on YARN.

I'd like propose the next stage of this effort: *Simplified and first-class support for services in YARN*.

This proposal is in essence a continuation of YARN-896 and a hop beyond. YARN-896 originally started (and continues) to be a JIRA where we track all the big and little things that are needed to get services working. In that sense, the line between YARN-896 and this JIRA is a bit blurry. The chief rationale for filing a separate new JIRA is threefold:

1. Do a fresh survey of all the things that are already implemented in the project
2. Weave a comprehensive story around what we further need and attempt to rally the community around a concrete end-goal, and
3. Additionally focus on functionality that YARN-896 and friends left for higher layers to take care of and see how much of that is better integrated into the YARN platform itself.

State of the art

YARN has so far restricted itself to having a very low-level resource-management API that can support any type of application. Application frameworks like Hadoop MapReduce ended up exposing higher level APIs that end-users can directly leverage to build stuff on top of YARN. On the services side, Apache Slider has done something similar.

Bringing services on YARN can happen in one of the following ways: (1) Use a framework on YARN (for e.g. Slider) that supports running the existing services *without* any modifications (2) Modify an existing service to also be aware of YARN (3) Write a *new* YARN application from scratch that leverages YARN APIs directly in order to run a long running service.

Relying on a framework (approach #1 above) has the advantage of having the ability to spawn services through specification (and without writing new *code),* hiding a lot of complex low-level details like state management, fault-tolerance etc, and of exposing simpler scheduling constructs and usage model to the user. Users/operators of existing services typically like to avoid approach #2. Developers of new services may want to go the new-app route (approach #3), trading off complexity for more control/flexibility.

Let’s take a concrete example of a service to illustrate the current state of the art and potential future direction: *Apache HBase* app running on YARN through the services framework Apache Slider. Though this is an example of approach #1 above, most of the arguments hold true for other approaches too.

In the beginning, we couldn’t keep an HBase app running for a long time on YARN due to issues like ApplicationMasters (AM) crashing enough times thereby causing YARN to fail the application, or tokens related to security expire after running for a while. A failed-over AM couldn’t connect to the old/running containers as YARN would kill all containers on AM-crash. Logs of services were never accessible centrally as log-aggregation only used to kick-in at application-finish. We also didn’t have ways to figure out the service-end points (inside each container) once an app started running. Services tend to have an affinity towards specific nodes (based on various constraints) and this was not possible due to lack of specific scheduling features in YARN. Through various efforts, we have solved many of these problems.

Despite our efforts so far, there are a gamut of things that we still cannot do - features that service users expect otherwise. We still don’t have a clean way of supporting services’ upgrades - a feature that isn’t so critical for short running apps (the current solution there being (a) running existing apps on older version and (b) only newer apps start pulling the newer bits). We don’t have good scheduling models when we try to run short running apps and services all on the same physical YARN cluster. Once you start running a service on YARN, methods to constantly monitor a service for faults and/or down-time are completely missing. Existing (and work-in-progress) metrics infrastructure is so tailored to short-running apps that services on YARN have to fend for themselves w.r.t metrics collection, propagation, and long-term storage.

Even on the side of developers, bringing a new service on YARN is not a simple experience. The APIs of existing frameworks are either too low-level (native YARN), require writing a lot of new code or writing a complex spec (for declarative frameworks). Packaging and installing a service forces developers to go through non-standard experiences like custom tar-balls. A lucid story around exposing end-points of services to users in a simple way is also lacking.

The other problem is a side-effect of YARN’s focus on limiting itself to low level APIs. Functionality like fault handling, basic configuration management ended up being implemented again and again in multiple frameworks like Slider, REEF, Twill etc. With this fresh look at services, we should try to reason, on a case-by-case basis, and absorb some very common functionality into YARN so that the ecosystem can benefit a richer out-of-the-box experience.

---

(2) Proposal Overview

In this section, I give an overview of our entire proposal. I'll further deep dive into each of the individual pieces in section (3) - please hold on to any questions you may have till then!

First-class support for services

- **Recognizing services: Special handling of preemption, container reservations etc.**: There are a few key areas where special recognition of services is not only unavoidable but very necessary. Preemption, reservations logic etc need to understand if an application has long running containers and make decisions accordingly.

- **Auto-restart of containers**: Long running containers may crash for various reasons and need to be restarted. For services, it will be good to have YARN automatically restart containers based on a policy.

- **Allocation reuse for application upgrades**: To support upgrades, we need a first class way of letting applications reuse the same resource-allocation from YARN for multiple launches of the processes.

- **Resource re-localization for reconfiguration / upgrades:** Once a service container starts, it may need to reinitialize itself with updated/new artifacts. YARN should enable applications to reuse an existing allocation but restart the container with a modified set of *LocalResources*.

- **Enabling dynamic configurations:** So far, we always let apps drive their configurations. Dynamic configurations can be a pain for apps. This calls for help from YARN w.r.t a few things. We need better ways of propagating important information (like log-dirs) down to the containers. Enabling additional ways to dynamically populate runtime configuration is very useful for apps. Other dynamic configuration items like service's endpoint information also needs to facilitated.

- **Network resource allocation:** Once you start running multiple instances of the same application (and of other applications) on the same node, YARN will have to coordinate the sharing of network resources needed by services. This means allocation of either ports or network addresses.

- **Scheduling:** YARN-896 covers a host of scheduling features that are useful for short-running applications and services alike. We should focus now on a baseline set without which running services on YARN becomes a hassle: updating our queue models in the presence of queues, enabling affinity, anti-affinity, gang-scheduling, malleable container sizes, resource-delegation and auto-spawning of system services.

- **Resource profiles:** With the addition of more and more resource-types like disk and network, specifying requirements of applications is getting cumbersome. Inappropriate specifications also lead to a lot of resource waste. YARN should have simpler ways of modeling resource-requirements: one of them being *resource-profiles* that are ready-for-use by apps.

- **Storage requirements for services:** YARN allocates *local-directories* to application for their local-storage needs. Many services are stateful and need help from YARN. This has implications on the life-cycle of such data, on how we tackle things like allocation / container-ids during container restarts, newer isolation requirements on disk usage given the long-life nature of services, and the ability to schedule disk/spindle resources better.

- **Service registry:** The YARN service registry today provides facilities for apps to register their endpoints and for clients to discover them. We need to first complete the present story - there are a few important tasks that still need addressing. We also need to simplify the access to the registry entries: we propose a solution to expose the registry information through a more generic and widely used discovery mechanism: DNS. Finally, solving the problem of new AMs discovering old containers can share some infrastructure that can help us scale to large number of registry-readers too.

Simplifying support for services

- **Framework**: With YARN restricting itself to having a very low-level API that can support any type of application, frameworks like Apache Hadoop MapReduce ended up exposing higher level APIs to facilitate building applications on top of YARN. On the services side, Apache Slider has done something similar. With our current attention on making services simplified, it’s time to take a fresh look at how we can make Apache Hadoop YARN support services well out of the box. We propose assimilating the client, ApplicationMaster etc of an existing framework like Apache Slider into YARN in order to serve this purpose.

- **API layer**: In addition to building critical building blocks inside YARN, we should also look to simplifying the user facing story of building services. Here again, our experiences building real-life services like HBase, Storm, Accumulo, Solr etc on top of Slider gave us some very good learnings on how simplified APIs for building services will look like. We have working prototypes of a simple services API layer backed by REST interfaces that we wish to contribute as part of this initiative. This layer acts as a single point of entry for creation and lifecycle management of services on YARN through Slider.

- **Packaging**: There are efforts underway to support newer packaging models like Docker in YARN. We can leverage simpler packaging of different container formats like Docker in simplifying services support on YARN. YARN should optionally support service definitions with containerized formats as first-class packaging specs: NMs should start recognizing container images as first-class artifacts and then interact with image stores in localizing such images and managing the lifecycle of the localized artifacts. Further, in order to make the docker integration seamless, YARN should support APIs for docker universe in a first-class fashion.

- **Monitoring:** YARN has always supported monitoring of containers' resource usage. It also inherits basic container liveliness monitoring for free from the underlying OSes. Beyond this basic resource and liveliness monitoring, we today do not have any mechanisms of monitoring containers' activities - an important need specifically for services. Historically, we've always left this as an exercise for application developers, but having a basic support of this at YARN itself will make the life of services and application developers simpler.

- **Metrics**: Via [YARN-2928] YARN Timeline Service: Next generation, we are building a key part of our metrics infrastructure for applications, but it is only amenable for applications that *finish* in a reasonable time period. Metrics collection for services calls for a different story: (a) we can leverage most of the YARN-2928 infrastructure and/or (b) dictate a first class way of letting users wire up their services to existing metrics systems (of the ilk of Ganglia) in a consistent way.

---

(3) Deep Dive

Before we deep-dive into the functionality that I think should be built, let us delve into the present state of things with services on YARN.

(3.1) State of the art

Here is my attempt at summarizing in reasonable detail of what we have accomplished so far through existing efforts in the YARN project.

(3.1.1) Fault tolerance of AMs of long running applications

Before this effort, there were only a limited number of AM retries possible - this logic makes sense for short-running apps as it is better to give up after a while instead of retrying for ever on potentially unrecoverable failures. Extending this to services, it's only a matter of time till a service gets killed by YARN due to exhausted AM retries.

Via YARN-611 Add an AM retry count reset window to YARN RM, we can now have ApplicationMasters run for a long time / forever. And through policy, YARN can (a) reasonably forget failures over time, along with (b) the ability to act on them when they cascade in a very short interval.

Other orthogonal efforts like (1) YARN-614 Separate AM failures from hardware failure or YARN error and do not count them to AM retry count and (2) YARN-2074 Preemption of AM containers shouldn't count towards AM failures also helped improve the experience of long running AMs.

(3.1.2) Work-preserving AM restart

As a result of this effort at YARN-1489, we added (1) a YARN API that tells RM to not kill all the containers of the previous *ApplicationAttempt* and (2) an API for the new *ApplicationAttempt* to know where the previously launched containers are running.

This enables running containers to happily chug along even if AM crashes and fails over. This is not so much of a deal for short-running apps, as they can simply retry outstanding work. For services, it is a deal-breaker as, by definition, service containers should survive platform AM restarts.

(3.1.3) Preliminary handling of logs of long-running services

Through YARN-2443 Handling logs of long-running services on YARN, we have added an early way of automatically aggregating rolled logs of services periodically. This ties in with the existing log aggregation mechanism used for containers of short-running apps.

The reason I called this preliminary is because it hardly compares to the experience one gets by running services outside of YARN - there is lot left to be done here, besides also letting users aggregate logs to the system of their choice.

(3.1.4) A preliminary service-registry facility

Through YARN-913 [Umbrella: Add a way to register long-lived services in a YARN cluster] we added a service-registry layer in YARN that lets any services running on YARN to register their endpoints (running inside containers) as queryable by clients. This is especially important for services in a dynamic environment like YARN, since we cannot pre-determine the hosts and/or ports where services will come up.

Today, we implemented service-registry as a client side java library backed by ZooKeeper based store underneath.

(3.1.5) Container placement via Node-Labels

Through (1) YARN-796 Allow for (admin) labels on nodes and resource-requests and (2) YARN-3214 Add non-exclusive node labels, we added the ability for administrators to specify labels against nodes and let applications specify their desires via the API.

We added the notion of partitions - node-labels that have an implication on queue capacities - and how they are shared - exclusive / non-exclusive, and are now extending it to have other types like constraints (YARN-3409 Add constraint node labels)

Labels and related placement features have a lot more significance when it comes to scheduling services. There is a need for lot more power and flexibility here.

(3.1.6) Rolling upgrades and Work-preserving RM and NM restarts

Through, YARN-666 [Umbrella] Support rolling upgrades in YARN and sister JIRAs, YARN now supports rolling upgrades and restarting RMs and NMs without killing the containers. Though these features are useful for short-running applications too, they are that much more important for services which tend to be flat-footed and don't particularly like getting killed all the time during upgrades.

Let us now jump into the proposal for the next stage of this effort: *Simplified and first-class support for services in YARN*.

(3.2) First-class support for services

Let's first look at what functions I believe are best supported in a *first-class* fashion at YARN level and without which services won't be able to live with. In the next major section, I'll cover functions that I believe are not specifically *required* to be supported in YARN but it is desirable to support them nevertheless in order to avoid duplicating them in every framework and to make services support much more accessible.

First up, NodeManager needs to start dealing with services better than it does today. Below, I discuss some of the important functionality that we need.

A general comment on what follows: In most of the topics, I try to draw an outline of *what* is needed and *how* we can accomplish it, but that isn't the case for all the topics. Where applicable, it is assumed generally that we will file sub-JIRAs where we can explore the topic much more expansively and arrive at a design.

I've also explicitly marked specific efforts with a special tag '**[Task]**' so as to highlight items that are defined clear enough to start working on them without much more deliberation, assuming consensus.

(3.2.1) Recognizing services: Special handling of preemption, container reservations etc.

Though it is desirable to not special-case services anywhere in YARN, there are a few key areas where such special recognition is not only unavoidable but very necessary. For example, preemption and reservation of long running containers have different implications from regular ones.

Preemption of resources in YARN today works by killing of containers. Obviously, preempting long running containers is different and costlier for the apps. For many long-lived applications, preemption by killing will likely be not even an option that they can tolerate.

[Task] Preemption also means that scheduler should avoid allocating long running containers on borrowed resources.

On the other hand, today's scheduler creates reservations when they cannot fit a container on a machine with free resources. When making such reservations for containers of long running services, the scheduler shouldn't queue the reservation behind other services running on a node - otherwise the reservation may get stuck unfulfilled forever.

[Task] Preemption and reservations logic thus need to understand if an application has long running containers and make decisions accordingly.

[Task] There is an existing JIRA YARN-1039 Add parameter for YARN resource requests to indicate "long lived" which was filed to address some of this special recognition of service containers. The options were between a boolean flag and a long representing the life-cycle, though in practice I think we will need to have both.

(3.2.2) Auto-restart of containers

Today, when a container (process-tree) dies, NodeManager assumes that the container's allocation is also expired, and reports accordingly to the ResourceManager which then releases the allocation. For service containers, this is undesirable in many cases. Long running containers may exit for various reasons, crash and need to restart but forcing them to go through the complete scheduling cycle, resource localization etc is both unnecessary and expensive. **[Task]** For services it will be good to have NodeManagers automatically restart containers. This looks a lot like inittab / daemon-tools at the system level.

We will need to enable app-specific policies (very similar to the handling of AM-restarts at YARN level) for restarting containers automatically but limit such restarts if a container dies too often in a short interval of time.

[YARN-3998] Add retry-times to let NM re-launch container when it fails to run is an existing ticket that looks at some if not all of this functionality.

(3.2.3) Allocation reuse for application upgrades

Once auto-restart of containers is taken care of, we need to address what I believe is the second most important reason for service containers to restart - upgrades. Once a service is running on YARN, the way container allocation-lifecycle works, any time the container exits, YARN will reclaim the resources. During an upgrade, with multitude of other applications running in the system, giving up and getting back resources allocated to the service is hard to manage. Things like Node-Labels in YARN help this cause but are not straight-forward to use to address the app-specific use-cases.

We need a first class way of letting application reuse the same resource-allocation for multiple launches of the processes inside the container. This is done by decoupling allocation lifecycle and the process life-cycle.

The JIRA YARN-1040 De-link container life cycle from the process and add ability to execute multiple processes in the same long-lived container initiated this conversation. We need two things here: (1) **[Task]** the ApplicationMaster should be able to use the same container-allocation and issue multiple **startContainer** requests to the NodeManager. (2) **[Task]** To support the upgrade of the ApplicationMaster itself, clients should be able to inform YARN to restart AM within the same allocation but with new bits.

The JIRAs YARN-3417 AM to be able to exit with a request saying "restart me with these (possibly updated) resource requirements and YARN-4470 Application Master in-place upgrade talk about the second task above but we also require, what folks at Apache Tez call as, Resource re-localization.

(3.2.4) Resource re-localization for reconfiguration / upgrades

Once a service container starts, it may need to reinitialize itself after getting either new configuration files or new binaries. Similar to what we mentioned in the preceding section, doing this without help from YARN is painful and expensive.

[Task] YARN should enable applications to reuse an existing allocation but restart the container with a modified set of LocalResources.

[Task] The additional effort beyond allocation-reuse for this to work is to change NodeManager life-cycle to be able to keep localizations going on throughout - this is very useful for the purpose of service reconfiguration. Further, by continuous localizations, we also unlock another useful functionality - being able to let containers download and use new jars, dictionaries and other resources while they are still running (Apache Tez relocalization).

(3.2.5) Enabling dynamic configurations

Services have configuration needs. So far, we always let applications drive their configurations. This works reasonably okay for static configurations that don't change at all and don't depend on environment, but there are a few gaps to be filled w.r.t dynamic configurations which will help services simplify their life-cycle. This includes help from YARN w.r.t a few things

- All of our platform-to-application communication currently is only through process environment variables: for e.g. *ApplicationContants.NM_HOST*. With things like Linux CGroups, containerization through docker etc, it is now possible to launch multi-process containers where the solution of environmental-variables breaks down. **[Task]** We need better ways of propagating important information down to the containers - information like container's resource size, local-dirs and log-dirs available for writing etc.

- Handling dynamic configuration: As mentioned earlier, most services depend on dynamic information like container resource size, local-dirs and log-dirs available which are not known till very late in the container launch cycle. Today apps that depend on configuration files need to figure out ways on their own to take this information from NodeManager and generate files. **[Task]** Enabling ways to dynamically populate some of this runtime configuration is very useful for apps. We cautiously limit ourselves to a small set of important use-cases to avoid slipping ourselves into building a large-scale configuration management system / templating engine, the likes of which already exist outside of YARN.

- One other type of dynamic configuration is service's endpoint information. For a complex app of multiple container types / components, some of the components depend on host / port information of where other components bound to. Such dynamic configuration also needs to facilitated, most likely in conjunction with Service Registry mechanism.

(3.2.6) Network resource allocation

Services use network resources. Once you start running multiple instances of the same application, or of other applications, on the same node, YARN will have to coordinate the allocation of network resources. Applications may either depend on a specific component being always located on a specific host, or being always bound to a known port irrespective of the host where it runs.

At the most basic level, there are two cases here: Disambiguate containers by (1) allocating ports to them through YARN or (2) have a thin networking layer and let each container have its own network address and/or private network with other containers of the same application.

Both of these need help from YARN and layers below. YARN will have to (a) schedule these resources per node, (b) schedule across applications in the cluster, (c) intermediate reservation of these resources by talking to layers below, and (d) figure out mechanisms of passing them down so that the services can use them in configuration, life-cycle etc.

(3.2.7) Scheduling

YARN-896 covers a host of scheduling features that are useful for short-running applications and services alike. Below, I list only a few very important features that we need to help schedule services better. Think of the following as a baseline set without which running services on YARN becomes a hassle.

- **Queue models**: YARN schedules applications grouped under the notion of a queue. Depending on the scheduling-policy, queues have their own configured capacities / shares etc. With services in the picture, with some of them potentially running for-ever, the scheduling model in users' minds is completely disrupted. Without services, users would assume that if they get in a queue, based on the queue-policy and based on how long previously queued applications in a queue run-for, they can rely on getting resources eventually. Outside of queues that are strictly planned, this assumption completely breaks down in the presence of one or more services. **[Task]** Our queue models should either surface these *updated available resources* better to the applications or evolve altogether to let users know that the used share by running-services is essentially taken away from the queue. For example, trying to run *adhoc* services without careful planning and coordination with other members of the queue can lead to livelocks/deadlocks that may need offline resolution.

- **Affinity:** Services desire their components to flock to other components in the same app, or to components in another peer-application. **[Task]** YARN should support various levels of affinity requirements in a first-class fashion. There are various dimensions to this: machine, rack, node-label / partition, hard or soft affinity etc.

- **Anti-affinity**: This type of placement is the one that we've seen most requested for Slider based services. The specific example is the need for avoiding physical nodes that are already running one copy of a service component. The way it has been implemented in Slider today is by explicitly requesting one at a time - a very slow process. **[Task]** We need first-class support for Anti-affinity in YARN. YARN-1042 add ability to specify affinity/anti-affinity in container requests is an existing JIRA that introduces both the scheduling request types.

- **Gang scheduling**: Some services depend on the availability of enough resources to start all their components at the same time - it's not useful to get a part of total resources unlike traditional MapReduce like applications. **[Task]** YARN should have a way of requesting resources in a gang. YARN-624 Support gang scheduling in the AM RM protocol covers this type of requirement. Gang scheduling may work very well with upfront reservation of resources (YARN-1051 YARN Admission Control/Planner: enhancing the resource allocation model with time) - requesting for a gang of resources *without* a previous reservation on them may result in unbounded waits.

- **Malleable container sizes**: Today's resource-management logic assumes that resources allocated to a container are fixed during its entire lifetime. For service containers that can run for a long time, it is desirable to change the size of the allocation based on the service's own internal demand/usage patterns. The only way to accomplish this today is by releasing the allocation and then requesting a new one with the needed size. **[Task]** YARN-1197 Support changing resources of an allocated container aims at facilitating run-time change of sizes.

- **Resource delegation**: There is a need for applications to be able to negotiate resources with YARN and delegate the obtained resources (instead of using them for themselves) to another service. This enables hybrid interaction of applications and services all running on the same cluster. **[Task]** YARN-1488 Allow containers to delegate resources to another container introduces this idea, a pattern commonly seen with data/query-services with which other YARN applications interact.

- **Auto-spawning of some (*system)* services**: In a typical cluster with services support, one can imagine a *default* set of services that are expected to start when the cluster is initialized/restarted and always run on the cluster. **[Task]** YARN should enable administrator to have a set of services to be automatically started and managed together with the base platform itself.

  YARN today already has a facility to start *admin-blessed* code along with NodeManager via *Auxiliary Services.* Auxiliary services are a weak mechanism with bad resource/software isolation implications and so are not an encouraged pathway for system services. In the medium term, we should definitely fork off Auxiliary services from the NodeManager address-space and may be even deprecate them. With true support for system services together with affinity features, we don't need auxiliary services as they exist today.

Further, YARN's Resource Request mechanism is powerful, but a host of new feature requirements, including some of the above, are making it increasingly more and more complex to understand and implement them within the code-base. Orthogonal to the services initiative, we have another upcoming proposal to generalize resource-placement in YARN so that we not only simplify our existing model, but also make it extensible enough to accommodate any future requests.

(3.2.8) Resource profiles

YARN has always had support for memory as a resource, inheriting it from Hadoop-(1.x) 's MapReduce platform. We also added support for CPU as a resource (YARN-2 / YARN-3). Currently, there are multiple efforts in progress to add support for various resource-types in YARN such as disk (YARN-2139), and network (YARN-2140).

As we started adding on more and more resource-types, two things became clear.

- One, understanding how to specify their resource-requirements to YARN has become a burden for users. Memory was relatively straight-forward - inform YARN about your peak memory requirements and off you go. With CPU, it became a little complicated - there is no single peak cpu requirement - but still manageable. Your app can work reasonably okay whether you have maximum amount of cpu or a certain minimum amount. Adding more resource-types like disk and network to the mix only makes the situation worse.

- Two, given the interaction between various resource-types, their availability on any node and varying needs from different application types, many sites end-up with a lot of internal and/or external fragmentation of resources.

In many systems outside of YARN, users are already accustomed to specifying their desired 'box' of requirements where each box comes with a predefined amount of each resources. Admins would define various available box-sizes (small, medium, large etc) and users would pick the ones they desire and everybody is happy. YARN-3926 Extend the YARN resource model for easier resource-type management and profiles aims to do the same in YARN via the introduction of Resource Profiles. This helps in two ways - one, the system can schedule applications better and two, it can perform intelligent over-subscription of resources where applicable.

Resource profiles are all the more important for services as (1) similar to short running apps, you don't want to fiddle with varying resource-requirements for each container type and (2) services usually end up planning for peak usages, leaving a lot of possibility of barren utilization.

(3.2.9) Storage requirements for services

Today, applications satisfy their local storage requirements with help from YARN. When an application starts, YARN allocates directories, called *local-directories*, where the containers can write their state or data. These directories can have different scopes: (a) *container-scoped* directories can hold data as long as the container lives (b) *application-scoped* directories serve as places where containers can write data so that other containers (which may start long after the writers) can access that data on that node.

Many services are stateful. While auto-restarts, allocation reuse etc take care of resources' state, services need help w.r.t data storage. There are two cases here:

A. *Services that have all their state on a persistent distributed storage and hence don't have local-storage requirements*: These are simple to run on YARN as there are no new feature-asks.

B. *Services that have local-storage requirements*: With these services, you don't want to lose their previously persisted state during container restarts, container upgrades, application upgrades etc. **[Task]** This calls for a clear story on the life-cycle of such data - how long does it live, what does it rely on if it not tied to the container life-cycle, if it is bound to the app, what happens to the data written by an old container on a node where this service is never going to go again etc. **[Task]** This also has implications on how we tackle things like allocation / container-ids during restart. For e.g., if the new instance of a component has a different container-id, then it will not have a straight-forward way to discover where the previously written data is located.

[Task] Within the context of docker-like containers, YARN should have ways to mount read-only, read-write directories into the containers to expose the same power available to non-containerized workloads via container-scoped/application-scoped local-directories.

[Task] Also given the long-life nature of services, local-storage needs will also mandate newer isolation requirements on disk usage. For short-running apps, it was useful but not necessary to put strict disk-capacity limits - in fact YARN never enforced this. But with services, some of which can run for-ever, giving access to raw-disks without controls on how much they can write can wreck havoc on other applications sharing a node.

In addition to resource-usage isolation, many services have specific needs for disk locality and/or exclusive-spindle access. This is specifically an important requirement for I/O-intensive services (for e.g. Apache Kafka). **[Task]** YARN should support better disk scheduling and locality. The JIRA YARN-2139 [Umbrella] Support for Disk as a Resource in YARN discusses some of these requirements.

(3.2.10) Service registry

(3.2.10.1) Complete the present story

The YARN service registry today provides facilities for applications to register their endpoints and for clients to discover them. There are a few important tasks that still need addressing

- The lifecycle of the service registry entries are supposed to be managed in a coordinated manner between the frameworks (for e.g *Apache Slider*) and the platform. There are a few things that still need some progress: (a) YARN-2571 RM to support YARN registry proposes performing some of these operations through ResourceManager (b) For some of the cleanup operations, we need support in YARN for what I call cleanup containers: YARN-2261 YARN should have a way to run post-application cleanup.

- We also need to straighten out the the user-side semantics around the service registry entries: **[Task]** when is a registry entry deterministically 'available for use' ? This is required for applications which have multiple components. For e.g., a new component in a specific application may not be able to launch unless the service entries for the dependent component are available and ready for use. A similar problem may impact a service when one or more of its dependent services fail and move(s) on restart - the DNS proposal below should help address these issues.

(3.2.10.2) Simplified discovery of services

In addition to completing the present story of service-registry, we also need to simplify the access to the registry entries. The existing read mechanisms of the YARN Service Registry are currently limited to a registry specific (java) API and a REST interface. In practice, this makes it very difficult for wiring up existing clients and services. For e.g, the dynamic configuration of dependent end-points we discussed in (3.2.5) is not easy to implement using the present registry-read mechanisms, without code-changes to existing services.

A good solution to this is to expose the registry information through a more generic and widely used discovery mechanism: DNS. Service Discovery via DNS uses the well-known DNS interfaces to browse the network for services. YARN-913 Umbrella: Add a way to register long-lived services in a YARN cluster in fact talked about such a DNS based mechanism but left it as a future task. **[Task]** Having the registry information exposed via DNS simplifies the life of services.

(3.2.10.3) Enabling discovery of AMs

Even after Work-preserving AM restart (Section 3.1.2), we still haven't solved the problem of old running containers not knowing where the new AM starts running after the previous AM crashes. This is a specifically important problem to be solved for long running services where we'd like to avoid killing service containers when AMs fail-over. So far, we left this as a task for the apps, but solving it in YARN is much desirable. **[Task]** This looks very much like service-registry, but for app-containers to discover their own AMs.

Combining this requirement (of any container being able to find their AM across fail-overs) with those of services (to be able to find through DNS where a service container is running) will put our registry scalability needs to be much higher than that of just service end-points. This calls for a more distributed solution for registry readers - something that is discussed in the comments section of YARN-1489 [Umbrella] Work-preserving ApplicationMaster restart and MAPREDUCE-6608 Work Preserving AM Restart for MapReduce.

In addition to discovery of AMs by the containers, there is a related problem of AMs understanding what *sort* of container were running *before* the fail-over. Here again, historically we left it up to the application - MapReduce AM can determine which task-type is running by looking at JobHistory, and Slider does an implicit mapping of component-type to container-priority (!). **[Task]** YARN should consider having a way of tagging containers so that AMs can quickly associate the container-types after fail-over.

(3.3) Simplified support for services

Like mentioned before, in this major section, I'll cover functions that I believe are not specifically *required* to be supported in YARN but it is desirable to support them nevertheless in order to make services support more accessible.

(3.3.1) Framework

Repeating what we discussed above, YARN has so far, by design, restricted itself to having a very low-level API that can support any type of application. Frameworks like Apache Hadoop MapReduce, Apache Tez, Apache Spark, Apache REEF, Apache Twill, Apache Helix and others ended up exposing higher level APIs that end-users can directly leverage to build their applications on top of YARN.

On the services side, Apache Slider has done something similar. Slider was initially proposed as a separate project to learn about modeling services on top of YARN in much faster development cycles than the Hadoop project. We believe it served its goals.

With our current attention on making services first-class and simplified, it's time to take a fresh look at how we can make Apache Hadoop YARN support services well out of the box. Beyond the functionality that I outlined in the previous section on how NodeManagers and Scheduler can be enhanced to help services, the biggest missing piece is the framework itself. There is a lot of very important functionality that a services' framework can own together with YARN in executing services end-to-end. Slider AM already handles a great deal of such functionality, and there's more to come.

Assimilating the client, ApplicationMaster etc of an existing framework like Apache Slider can serve this purpose really well. My early informal discussions about this with few Hadoop and most of the Slider community members yielded generally favourable feedback. The concrete proposal will be around taking Slider from incubation into Apache Hadoop where we can continue its evolution and development. Once there is general consensus here, we plan to initiate a more elaborate community discussion on how we can leverage the Slider project inside

Apache Hadoop project, with the explicit goal of enabling a great out-of-the-box services experience on YARN.

(3.3.2) API layer

In addition to building critical building blocks inside YARN, we should also look to simplifying the user facing story of building services. Here again, our experience building real-life services like HBase, Storm, Accumulo, Solr etc on top of Slider gave us some very good learnings on how simplified APIs for building services will look like.

We have offline working prototypes of a simple services API layer backed by REST interfaces that we wish to contribute as part of this initiative. The REST layer acts as a single point of entry for creation and lifecycle management of YARN services, dealing with YARN through Slider. Services here can range from simple single-component apps to the most complex, multi-component applications needing special orchestration needs.

We also have tentative plans to make this a unified REST based entry point for other important features like resource-profile management, package-definition lifecycle-management and service-discovery. We also need to flesh out its relation to our present much-lower level REST APIs in YARN for application-submission and management.

(3.3.3) Packaging

YARN today already supports specifying and auto-distributing arbitrary artifacts (*LocalResources*) needed in order to launch applications. Typical examples of this include java jar-files for Hadoop MapReduce, platform specific binaries for MapReduce Streaming, Pipes, any zip / tar files needed by the application / service etc.

This per-application packaging though flexible tends to be cumbersome for reasonably big applications and services that have lots of artifacts, with side-effects like difficult software isolation. Incidentally, there is a big effort underway to support newer packaging models like Docker in YARN - [YARN-2466] Umbrella issue for Yarn launched Docker Containers and YARN-3611 Support Docker Containers In LinuxContainerExecutor.

With the docker effort underway, we can simply run services on YARN through simpler packaging of different container-formats like Docker. **[Task]** YARN should optionally support service definitions with containerized formats as first-class packaging specs. What this internally also means from YARN's perspective is earning the ability for NMs to start recognizing container images as first-class artifacts and then interacting with image stores (docker registry) in (a) **[Task]** localizing such images (like today's *LocalResources: YARN-2479* DockerContainerExecutor must support handling of distributed cache) and (b) managing the lifecycle of the localized artifacts.

Today, most of the input to the *DockerContainerExecutor* are based on brittle environment variables. **[Task]** In order to make the docker integration seamless, YARN should support APIs for docker universe in a first-class fashion. **[Task]** There are also a bunch of issues around security and user-mapping in the context of Docker that need to be addressed: YARN-2480 DockerContainerExecutor must support user namespaces and YARN-3291 DockerContainerExecutor should run as a non-root user inside the container.

(3.3.4) Monitoring

YARN has always supported monitoring of containers' resource usage and has the ability to shoot them down if they go over the promised resource usage. YARN also inherits basic container liveliness monitoring for free from the underlying OSes.

Beyond this basic resource and liveliness monitoring, we today do not have any mechanisms of monitoring containers' activities - an important need specifically for services. Neither does YARN itself make or help AMs make decisions based on them. Historically, we've always left this as an exercise for application developers, but having a basic support of this at YARN itself will make the life of services and application developers simpler.

[Task] With services specifically, there is a generic need for a basic ability to periodically monitor containers on various dimensions (for e.g. RPC latency, HTTP response time etc), surface any discrepancies on any of these dimensions to the end-user and help AMs make decisions based on such status of the containers. This concretely can mean supporting basic service liveness checks of AM and containers by enabling the following

- Letting applications submit to the NodeManagers some sort of monitoring plugins as part of a service container launch. The plugin can be a simple command-line (similar to *ContainerLaunchContext)* with a few out-of-the-box plugins like an HTTP curl call.

- Having the NMs periodically invoke the plugin (ping the HTTP port of a container or an RPC endpoint), monitor the result and response time of the plugin execution, optionally remember a window of such values and report any spikes to the AMs.

- Services should also be able to define few alerts that YARN can then integrate into external alerting systems (of the ilk of Nagios). This obviously means that YARN will need some basic interfaces (like jmx) to be implemented by the applications.

- YARN should optionally help side-step all of this basic support and let the integration of applications directly into existing monitoring infrastructures that sites may have.

As you can see, the mechanics of monitoring is largely an open item that needs more fleshing out in its own design discussion. Some of the alerts may indeed depend on metrics that we collect per service - our next topic.

(3.3.5) Metrics

Via [YARN-2928] YARN Timeline Service: Next generation, we are building a key part of our metrics collection infrastructure for applications. However, as its design stands, we have made several key decisions there that make it only amenable for applications that *finish* in a reasonable time period. As an example, we specifically tried to avoid building time-series data-points for a single container because it results in untenable storage requirements for a large-scale cluster running hundreds of thousands of short running applications per day. But that is precisely the kind of information that is much needed for long running containers. Metrics collection for services thus calls for a different story.

[Task] There are a couple of options in sketching this out: (a) We can leverage most of the infrastructure of YARN-2928 for collection, storage and retrieval of service-containers. All we will need to do differently are metrics aggregation mechanisms and table design. (b) There are lots of existing metrics systems (of the ilk of Ganglia) that we can and should leverage. What YARN can do here is dictate a first class way of letting users wire up their services to metrics systems in a consistent way.

[Task] Another key part of handling metrics is YARN-3332 [Umbrella] Unified Resource Statistics Collection per node. Once we can collect statistics per container per node, we can have them routed to appropriate metrics systems. YARN-3332 will be specifically important if we go with option (a) above to build it together with YARN-2928.

[Task] One related addition that we need to have at YARN-3332 is the ability for service containers launched by YARN to register/deregister themselves to the “Per-node Statistics agent” and send metrics there so that YARN can (a) route them beyond and (b) act on some of them as needed. For example, some of the monitoring/alerting mechanisms in the preceding section may depend on the values of a specific container-metric.

---

(4) More topics for deliberation

In addition to the myriad of functionalities we have discussed so far, there are more topics that deserve further deliberation. In this section, I’ll try listing a few of those that we gave thought to, but didn’t take them to their logical conclusions so as to show a clear path of progress.

First up - **handling of logs**, YARN has an existing story on how we handle logs of containers - *the log-aggregation* feature. Log-aggregation is a service that runs on every NodeManager, listens for container-start / container-finish and application-start / application-finish events, monitors log-directories of containers and uploads the per-application aggregated logs to a FileSystem (HDFS), from where a user can easily access his/her logs. It has been successful in addressing the logs-accessibility problem that plagued Hadoop-1.x, but on the way ran into a few severe roadblocks, some of which are documented at YARN-431 [Umbrella] Complete/Stabilize YARN application log-handling. Beyond that, we've implemented a few things to reuse the same infrastructure for long-running services: YARN-2443 Handling logs of long-running services on YARN, though it is still needs more attention to make it seamlessly work. **[Task]** We should figure out how log-aggregation fits into handling of logs of services. In addition, there is a plethora of existing log management tools that can help. **[Task]** YARN should have ways to optionally integrate into existing log tools to help with activities like filtering, searching through, archiving logs.

Secondly, **the security story** of services on YARN is an evolving one. YARN has time-tested security mechanisms that it largely inherits from rest of the Hadoop. Existing design of Hadoop security (based off Kerberos) doesn't have special provisions for long running services. Within YARN, we have done a few things to take care of things like resource-localization, log-aggregation by delegating this responsibility to YARN itself: YARN-2704. Outside of this, services need to interact with YARN as well as other systems; and for both of this, we so far can only rely on services using keytabs. Existing long-running services, outside the context of YARN, already use keytabs to interact with a secure cluster. So, it was only a natural expectation to have these services continue to use keytabs once they start running under YARN. This may or may not fly at all the sites, depending on the specific security requirements, and on the ability to let arbitrary users create and distribute keytabs to various nodes through YARN. This needs more deliberation.

---

(5) Conclusion

Assuming we accomplish a bulk of what we proposed in this document, we can go back to our early example of running an HBase app on YARN. We will now have a clean way of supporting services' upgrades in YARN, powerful scheduling features to run an arbitrary service. Once a service is deployed on YARN, it can leverage the basic framework for monitoring services and viewing their metrics. For developers or services, with the simplified REST API, bringing a new service on YARN should be a simple experience, without a need for writing any new code. Standard packaging mechanisms like Docker can be used to develop services on a laptop and then use the same images to run on a large cluster. With DNS based registry, multiple apps and services can interact each other in a simple way.

To conclude, we put forward in this document a renewed proposal for services on YARN as a continuation of YARN-896. We make no claim to this being a comprehensive document about all-things-services, but it should serve as a *minimal set* needed. We hope that the document gives a fresh view of where things stand, and what we as a community can rally around and make services on YARN a simplified and first-class experience.
