Spark on Kubernetes Design Proposal

SPARK-18278

Andrew Ash (Palantir), Anirudh Ramanathan (Google), Eric Tune (Google), Erik Erlandson (Redhat), Matt Cheah (Palantir), Ilan Filonenko (Bloomberg), Sean Suchter (Pepperdata)

Last Updated: August 11, 2017

Background and Motivation

Containerization and cluster management technologies are constantly evolving in the cluster computing world. Apache Spark currently implements support for Apache Hadoop YARN and Apache Mesos, in addition to providing its own standalone cluster manager. In 2014, Google announced development of Kubernetes which has its own unique feature set and differentiates itself from YARN and Mesos[1]. Between 2014 and 2016, it has seen contributions from over 800 contributors with over 30,000 commits[2]. Kubernetes has cemented itself as a core player in the cluster computing world, and cloud-computing providers such as Google Container Engine, Google Compute Engine, Amazon Web Services, and Microsoft Azure support running Kubernetes clusters[3].

This document outlines a proposal for integrating Apache Spark with Kubernetes in a first class way, adding Kubernetes to the list of cluster managers that Spark can be used with. Doing so would allow users to share their computing resources and containerization framework between their existing applications on Kubernetes and their computational Spark jobs and applications. Although there is existing support for running a Spark standalone cluster on Kubernetes[4], there are still major advantages and significant interest in having native execution support[5]. The proposed design is then compared to existing work to show that the new architecture is beneficial in spite of what is currently available. This proposal concludes by highlighting some matters that are open to discussion.

High Level Overview

The high level idea is to integrate Kubernetes as a cluster scheduler backend within Spark. This enables Spark to directly be aware of Kubernetes in a manner similar to its awareness of Mesos, YARN and Standalone clusters. Spark-submit is made aware of new options that are used to inject the right parameters and create the driver and executor within Kubernetes that carry out the computation associated with the Spark Job.

This integration between Spark and Kubernetes allows both components to interact with each other intelligently. For instance, it allows Spark to leverage Kubernetes constructs that facilitate authentication, authorization, resource allocation, isolation and cluster administration. Similarly, framework level events in the cluster orchestration layer can be interpreted appropriately by Spark. Furthermore, features like CustomResources can be used to extend the Kubernetes API to control Spark jobs using the Kubernetes control plane.

Design

The general idea is to run Spark drivers and executors inside Kubernetes Pods. Pods are a co-located and co-scheduled group of one or more containers run in a shared context. The main component is `KubernetesClusterSchedulerBackend`, an implementation of `CoarseGrainedSchedulerBackend`, which manages allocating and destroying executors via the Kubernetes API. There are auxiliary and optional components: `ResourceStagingServer` and `KubernetesExternalShuffleService`, which serve specific purposes described further below.

Scheduler Backend

The scheduler backend is invoked in the driver associated with a particular job. The driver may run outside the cluster (client mode) or within (cluster mode). The scheduler backend manages pods for each executor. The executor code is running within a Kubernetes pod, but remains unmodified and unaware of the orchestration layer. When a job is running, the scheduler backend configures and creates executor pods with the following properties:

- The pod's container runs a pre-built Docker image containing a Spark distribution (with Kubernetes integration) and invokes the Java runtime with the CoarseGrainedExecutorBackend main class.
- The scheduler backend specifies environment variables on the executor pod to configure its runtime, particularly for its JVM options, number of cores, heap size, and the driver's hostname.
- The executor container has resource limits and requests that are set in accordance to the resource limits specified in the Spark configuration (`spark.executor.cores` and `spark.executor.memory` in the application's `SparkConf`)
- The executor pods may also be launched into a particular Kubernetes namespace, or target a particular subset of nodes in the Kubernetes cluster, based on the Spark configuration supplied.

Requesting Executors

Spark requests for new executors through the `doRequestTotalExecutors(numExecutors: Int)` method. The scheduler backend keeps track of the request made by Spark core for the number of executors.

A separate `kubernetes-pod-allocator` thread handles the creation of new executor pods with appropriate throttling and monitoring. This indirection is required because the Kubernetes API Server accepts requests for new executor pods optimistically, with the anticipation of being able to eventually run them. However, it is undesirable to have a very large number of pods that cannot be scheduled and stay pending within the cluster. Hence, the `kubernetes-pod-allocator` uses the Kubernetes API to make a decision to submit new requests for executors based on whether previous pod creation requests have completed. This gives us control over how fast a job scales up (which can be configured), and helps prevent Spark jobs from DOS-ing the Kubernetes API server with pod creation requests.

Destroying Executors

Spark requests deletion of executors through the `doKillExecutors(executorIds: List[String])` method.

The inverse behavior is required in the implementation of `doKillExecutors()`. When the executor allocation manager desires to remove executors from the application, the scheduler should find the pods that are running the appropriate executors, and tell the API server to stop these pods. It's worth noting that this code does not have to decide on the executors that should be removed. When `doKillExecutors()` is called, the executors that are to be removed have already been selected by the `CoarseGrainedSchedulerBackend` and `ExecutorAllocationManager`.

Resource Resolution

We plan on supporting three methods of resource resolution:

1. A single docker image containing both the application resources and the Spark distribution.
2. Specifying a URL (http://, s3:// or hdfs://) that is accessible to the cluster and having Spark download the dependencies or files.
3. Uploading locally hosted compiled code to the driver and executor pods.

Resources in Docker Images

This is a straightforward workflow which would require building an image containing all the necessary application resources and input files.

Many applications in the kubernetes ecosystem are designed to run entirely from a docker container, and have application code baked into the docker container. The spark-submit API should support directly running an application in a docker container that's hosted in a docker registry.[1]

Resources as URLs

Some dependencies may not be known at image build time, or may be too large to be shipped within a container image. Some workflows may use “standard” images and supply dependencies and input files at runtime. We must support the two following methods of resource resolution to support these use cases.

Resources specified as URLs will be localized by utilizing a Kubernetes feature known as init-containers. Init-containers are like constructors in the Kubernetes sense; they are containers which run to completion before the main container is started, and typically used to perform operations required for the main container to run.

A resource localizing init-container would be tasked with localizing resources specified as URLs. This clear separation between resource localization and the actual executor backend code allows for a clean separation of concerns.

Resource Staging Server

The spark-submit API allows users to run **locally-hosted** compiled code. This is achieved in the Kubernetes case using the Resource Staging Server (RSS). The RSS would be a completely separate standalone component that can be launched into the cluster.

Spark-submit would have knowledge necessary to upload said dependency from the local machine to the RSS. Following this, driver and executor pods would utilize the flow in (2), using init-containers to fetch the uploaded resources from the staging server to the individual driver and executor pods.

External Shuffle Service

The `KubernetesExternalShuffleService` was added to allow Spark to use Dynamic Allocation Mode when running in Kubernetes. The shuffle service is responsible for persisting shuffle files beyond the lifetime of the executors, allowing the number of executors to scale up and down without losing computation.

One possible implementation is as a DaemonSet that runs a shuffle-service pod on each node. Shuffle-service pods and executors pods that land on the same node share disk using hostpath volumes. Spark requires that each executor must know the IP address of the shuffle-service pod that shares disk with it.

The user specifies the shuffle service pods they want executors of a particular SparkJob to use through two new properties:

- `spark.kubernetes.shuffle.service.labels`
- `spark.kubernetes.shuffle.namespace`

`KubernetesClusterSchedulerBackend` is aware of shuffle service pods and the node corresponding to them in a particular namespace. It uses this data to configure the executor pods to connect with the shuffle services that are co-located with them on the same node.

There is additional logic in the `KubernetesExternalShuffleService` to watch the Kubernetes API, detect failures, and proactive cleanup files in those cases.

This is described in further detail in the Spark on Kubernetes: External Shuffle Service Design Proposal.

Kubernetes Interconnects

Spark Configuration

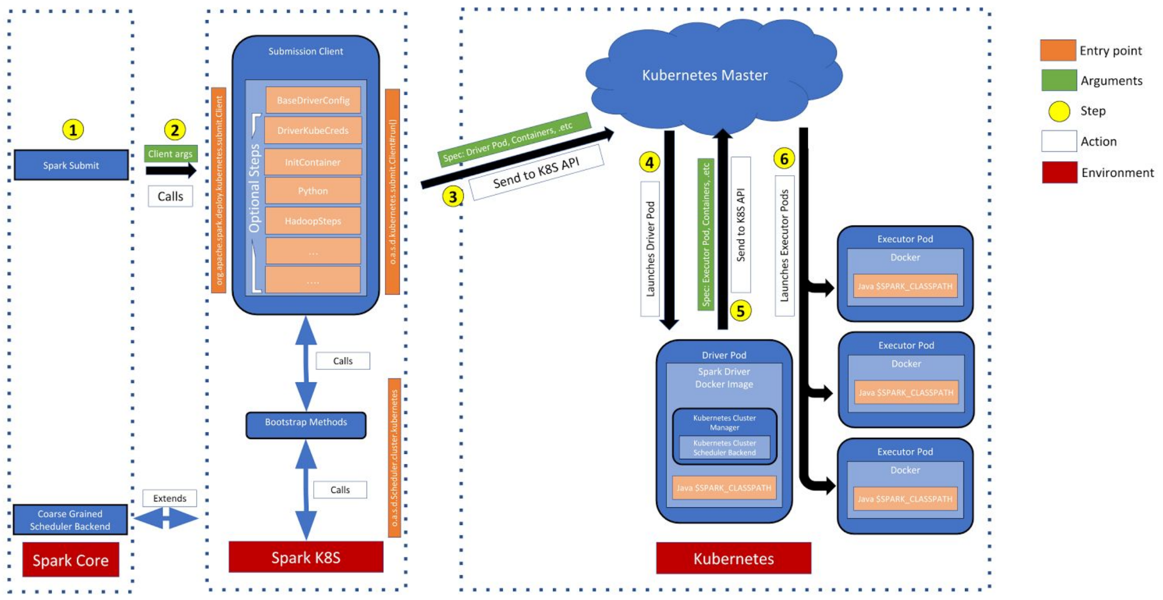
Several new Spark configurations can be supported for closer integration with Kubernetes primitives. These include namespaces, labels, annotations, node selectors, authentication/authorization options, and service accounts. These configurations are transformed appropriately into Kubernetes primitives that are applied on driver and executor pods.For an exhaustive list, please refer to link.

CustomResources

There is a need for a way for users to list, monitor status, and manage their spark jobs. Other cluster managers provide an interface to view all of a user's spark-jobs in a cluster, along with their state (queued, running, completed, failed, killed, etc.). As the spark framework would perform the entire function of a Kubernetes controller, Kubernetes does not have sufficient visibility into the Spark job. It is not possible to expose all the required detail about every spark job through the pods and their states.

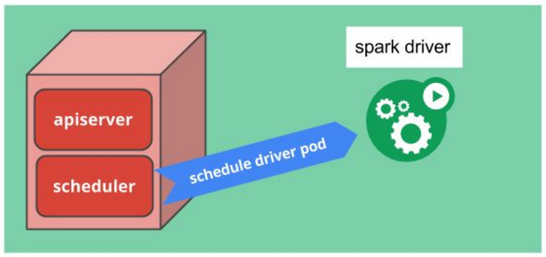
CustomResource objects (beta in Kubernetes 1.7) would allow for a first class representation of SparkJobs, allowing them also to be viewed and managed in rich UIs and through Kubernetes command line tools.

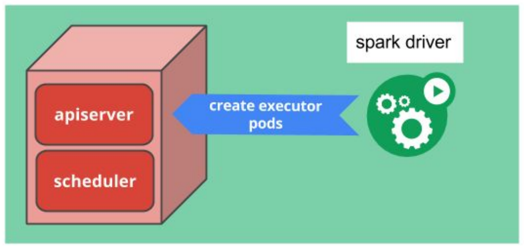
Summary Architecture Diagram

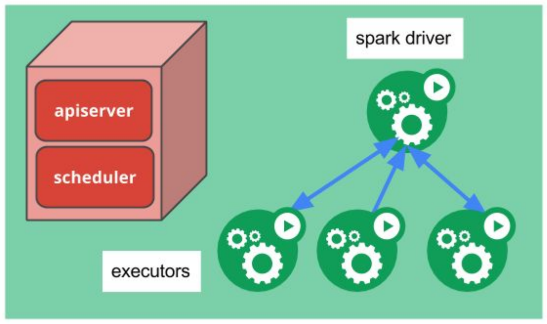


The above diagram outlines the general flow for spark jobs launched using Spark on Kubernetes. The diagram omits the staging of files and the role played by the optional resource staging server component (RSS). It summarizes the steps from the submission of a Spark Job to the corresponding driver and executor pods being created on the cluster.

The below figures go into a bit more detail of the Kubernetes components that a Spark driver pod interacts with. A new driver pod is scheduled, which in turn requests the creation of more executor pods, which then carry out the desired computation.

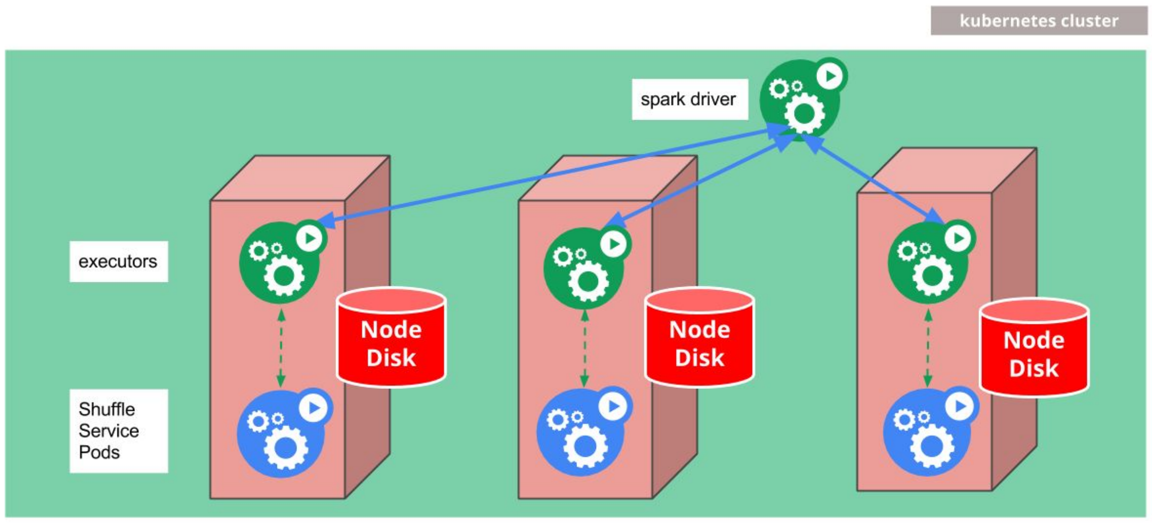






The below figure shows a **single Spark Job** with dynamic scaling turned on. Each executor shares disk with the shuffle service pods, allowing the shuffle service pods (which are long lived) to server files even after the executors are lost.

The diagram is simplified, but in a cluster, there may be more than one executor on a node, and they may belong to the same or different concurrently running Spark jobs.



Comparison with Spark Standalone on Kubernetes

If this feature is to be maintained in the long term, it is important to consider its benefits for users and if it is filling a need that other solutions do not satisfy. Some efforts have been made to integrate Spark with Kubernetes already. Kubernetes already provides an example for running a standalone Spark cluster on a Kubernetes cluster[4].

The proposed design is more flexible than the basic Kubernetes Spark example for several reasons:

1. Spark executors can be elastic depending on job demands, since spark can speak directly to kubernetes in order to provision executors as required.
2. The proposed solution allows leveraging of Kubernetes constructs like ResourceQuota, namespaces, audit logging, etc. Contrasting with Spark standalone where Spark's notion of identity is disparate and unaware of Kubernetes constructs like service accounts.
3. The proposed solution allows for greater isolation between Spark Jobs than standalone mode, as each of them runs in their own container, utilizing cgroups to enforce CPU and memory requests and limits.
4. When submitting jobs against a standalone Spark cluster running on Kubernetes, one is restricted to using the Docker image that the standalone Spark cluster was launched with to run the executors. The proposed design allows the user to specify a **custom docker image** for the driver and executors when the user submits the job.
5. The proposed design simplifies the process of running Spark jobs: while the current solution requires two steps to first run the standalone Spark cluster and then to run the Spark jobs against that cluster, the proposed design requires **no prior k8s setup** to run the user's application.
6. The proposed solution adds a Resource Staging Server (RSS) component that simplifies the deployment of user-specified jars and files which is still difficult with a Spark Standalone cluster on Kubernetes.
7. In the existing solution, resource negotiation for spark jobs is tied to the configuration of the standalone Spark cluster as well as the configuration of the Kubernetes cluster. If the second layer of the standalone cluster manager is eliminated, then the user can **unify all resource administration and configuration** into the Kubernetes framework alone.
8. Kubernetes is a popular container orchestration environment with wide adoption. Users who are already deploying applications on Kubernetes, or who plan to, will benefit from the ability to run Spark applications on Kubernetes as well, instead of running a separate standalone Spark cluster or managing a separate orchestration environment.

Implementation Plan

Because this is a large feature, and code reviewing more smaller changes is easier than fewer large changes, implementation will proceed incrementally in phases. We will try to coordinate these phases with the Spark release cycle so that phases are completing when the Spark merge window closes.

Additionally we intend to make phase one as small as possible so it is easier to get merged into apache/spark. Once that base is in many of the later phases can be worked on in parallel where possible so community members motivated by different features can work independently on those features.

Phase One: Static Allocation MVP

This is the phase to provide the MVP. It will have significant feature and security gaps but after completion should allow running a Spark job in k8s for a narrow use case.

- Spark-submit support for cluster mode
- Providing user code from both the client's local disk and remote locations
- Static number of executors
- Only Java + Scala support

This is complete.

Phase Two: Dynamic Allocation

- Dynamic allocation support
- External shuffle service prototypes, both with the sidecar approach and the daemon set approach. Assess the two implementations, and decide between them.

This is complete.

Phase Three: Complete Core Spark Features

- Support for remaining language bindings (Python, R)
- Spark must be able to talk to secure HDFS (design doc)
- Use K8s secrets to secure external shuffle service communication
- "Decent security" for data processed in Spark
    - Shuffle data protected at rest from neighbor processes

In progress.

Phase Four: Future K8s Features

This is the phase where the spark-k8s integration begins using k8s features that aren't yet released as of this writing (early Dec 2016).

- Job Management UI (similar to YARN's ResourceManager scheduler view)
    - May require integration with k8s Third Party Resources
- Shuffle Service Finalization
    - Potentially using a k8s local storage feature[6]
- Isolation (malicious jobs can't DOS neighbor jobs)
    - Fair sharing / queueing mechanism
    - Protection against disk exhaustion

    - Protection against deadlock with all cluster resources running drivers and none running executors
- Spark-shell / client mode
    - Potentially needs networking features in k8s

Open Questions

- Should we publish one docker image that has support for all of the language runtimes, or should we publish smaller docker images that each support a different language binding?
- How do we publish official docker containers? Is it sufficient to have docker files and instructions to build images?
- What are the performance implications of running executors within containers? Do we have benchmarks from other cluster scheduler backends to compare to?
- How can executor pods communicate with a driver running in client-mode when the driver is not in the cluster?
- What steps need to be taken to make the HistoryServer work well?

References

1. http://kubernetes.io/
2. http://blog.kubernetes.io/2016/07/five-days-of-kubernetes-1.3.html
3. http://kubernetes.io/docs/getting-started-guides/binary_release/
4. https://github.com/kubernetes/kubernetes/tree/master/examples/spark
5. https://github.com/kubernetes/kubernetes/issues/34377 - spark-k8s support
6. https://github.com/kubernetes/kubernetes/issues/7562 - local persistent storage

In-progress work

7. https://github.com/apache-spark-on-k8s/spark/
8. https://issues.apache.org/jira/browse/SPARK-18278

Footnotes

1. There are sometimes performance concerns around the efficiency of resource localization, especially when using on-prem clusters, where several executors may try to access and acquire resources from one or a few sources. One strategy to mitigate this includes replicated container image registries, and setting imagePullPolicy to ‘IfNotPresent’, ensuring that the docker image is fetched only once per node.