SPARK-23153: Support application dependencies in submission client's local file system

The Problem

We would like to support a common scenario where the user writes his code, builds an application jar and spark submit takes care of uploading the file in the cluster and runs the Spark Job.
Let's see how this is previously done with several resource managers.
With Spark on Yarn client files are stored/uploaded in the HDFS based distributed cache and so they are always accessible to the driver/executors.
On Spark on Mesos the fetcher is used by default to get the dependencies from a remote url. It is possible to integrate the fetcher with HDFS, S3 etc but it is not used by default as it requires to distribute hadoop files on all mesos nodes. The fetcher is an internal process of each mesos agent, running on every host. This concept is not K8s idiomatic and in general does not match the purpose of kubelets on K8s.
In Spark on K8s (version 2.3) there was a resource staging server that was removed due to performance, scalability and security issues and due to objections from the Spark community for its use.
In any case, in order to support this use, the core idea is to store a file somewhere and make it accessible for download in the cluster at runtime.

The current alternative approach to this idea, which is pretty common from a devops perspective when using container orchestrators like K8s, is to build your dependencies in your image and then deploy your application.
This requires to add your jar to an image, build the image and push the image to a registry from which you can pull the new image at runtime in the cluster.
In some use cases, as previously discussed here, it is also desirable to be able just download or inject the dependencies on the fly when the container is run for other reasons that go beyond the problem discussed in this context.
This might be the case for example due to security reasons where the only available images are some scanned basic ones.
That means building an image with dependencies at the client side or local fs is not realistic in some scenarios and makes several assumptions.

Let's now discuss available solutions to the problem of staging files originating from the client's fs at submission time.

Solutions

- Distributed Object Stores
  An example here is minio.

  Pros

  Can be used on jvm through its client to directly put dependencies in the cluster.
  Can be used via Spark Submit to upload dependencies.

  Cons

  Assumes there is a minio installation in the cluster.

- HDFS, GlusterFS, S3, GS etc
  Spark hadoop libraries allow to access any DFS for which there is hadoop API compatibility.

  Pros.

  It could integrated directly to the spark submit process. It is the most generic approach that can be implemented by reusing parts of the Spark code base.

  Cons

  Requires hadoop configuration files to exist at the side client, e.g. you need to pass credentials and somehow add these dynamically on the classpath at the spark submit side.

- Spark's existing internal file serving

  It is initialized via the RpcEnv and it supports (uses netty) :
  * - "/files": a flat list of files; used as the backend for [[SparkContext.addFile]].
  * - "/jars": a flat list of files; used as the backend for [[SparkContext.addJar]].
  * - arbitrary directories; all files under the directory become available through the manager,
  * respecting the directory's hierarchy.

  It is used for serving files to other processes like executors.

Pros
Exists anyway with any spark job submission.

Cons

Not persistent.
No artifact sharing between jobs.
Requires communication from the Spark Submit side directly to the driver pod to upload the dependencies and synchronization in case we need to start the application and we need to wait for an uploaded dependency.
Security concerns as it requires authentication/encryption from client to the file server.
Requires a connection between the client and the driver that goes through ingress.

- S2i
  Source-to-image (S2i) is a technique for making an application image using a builder image and the application's source code.
  Usually it assumes that there is a CI/CD infrastructure (eg. jenkins) that supports the whole process although not required.
  At minimum the flow is as follows:
    - Download the S2I scripts (or use the one from the inside builder image).
    - Download the application source.
    - S2I then streams the scripts and application sources into the builder image container.
    - It then runs the assemble script, which is defined in the builder image.
    - Save the final image.

  It is the major technique for building applications on Openshift. Also a similar tool jib can be found here.

  The builder image is assumed to have all the required libs to build the application code and create the final image.

  Pros

  - People can share base images for building applications.
  - Build is faster as there is no need for additional layers.

  Cons

  - Cannot be easily integrated with the Spark Submit code. It could be used as an external process to Spark Submit. That would require code to be accessible eg. github and store there all related dependencies.
  - It is a two-step process outside spark submit.

- Mounting PVs

  This requires to create a PV with the required dependencies which will be mounted at

container's runtime.

The major disadvantage is that you need to administer the PVs and pre-populate the volume with the dependencies. Also requires adding the jars under the volume mount path to the classpath which may be non-trivial.
So its a two-step process.

- Mounting Cloud Store buckets as fs.
  A more exotic technique would be to have a side-car container mounting an S3 bucket in the container's local fs. For more check here.
  This way the user could previously push his dependencies using the cloud storage sdk.
  The same concept could be applied with Google storage:
  https://github.com/s3fs-fuse/s3fs-fuse/wiki/Google-Cloud-Storage

  Note: The most obvious approach is to directly reference an S3 url via the Spark hadoop API, but it seems interesting so I captured it here as it does not need any hadoop config files.

  Cons
  - Its a two-step process outside spark submit
  - Limited to specific storage types.

- Sparckctl and the Spark Operator

  Sparkctl is enhanced cli tool used to manage Spark Jobs via the Spark Operator.
  We could enhance the tool to use the Spark Operator as the file server for dependencies.

  Pros
  We hide effectively the details of dependency management. Operator as a proxy for job submission handles all the details.

  Cons
  It is limited to the people using the spark operator.

Rejected Alternatives

Use the Api server and Config Maps to store artifacts

Although with the K8s 1.10 version ConfigMap objects support binary data via a new binaryData field we cannot use it for storing application dependencies.
All K8s api resource objects are stored in etcd which comes with several limitations:
https://github.com/etcd-io/etcd/blob/master/Documentation/dev-guide/limit.md#request-size-limit
Documentation/dev-guide/limit.md

The recovery time is the main reason though for tis restriction as you want to keeps your data limited in size:
https://groups.google.com/forum/#!topic/etcd-dev/vCeSLBKC_M8