External Volume Support via CSI

Background

Currently DC/OS already supports CSI volumes via DSS (DC/OS Storage Service), however there are some limitations about DSS:

1. DSS has some code specific for two homegrown CSI plugins (csidevices and csilvm), so it can only work with those two plugins but not any other 3rd party CSI plugins.
2. To work with DSS, a CSI plugin needs to implement GetCapacity / ListVolumes APIs so that the external storage can be modeled as Mesos raw disk resources and then offered to DSS. However there are a lot of 3rd party CSI plugins that do not implement them.

So we need a more generic way to support 3rd party CSI plugins so that DC/OS can work with broader external storage ecosystem and we'll benefit from continued development of the community CSI plugins. In the meantime, we want to support "data mobility", i.e. the ability to attach the same volume to multiple nodes in the cluster, so even if a node is down, the data will not be lost. This will help enable non-HA applications to run easily in a cluster across failovers, backed by external storage they can recover on another node with their previous state.

Goal

1. A generic way to support 3rd party CSI plugins in UCR.
2. Data mobility.

As MVP, we only support shareable volumes (i.e., a single volume can be attached to multiple nodes simultaneously, e.g., NFS, Portworx, but not EBS), in this way, even if a node fails, we can still reschedule the tasks onto another healthy node and attach the volumes to that node without having to detach the volumes from the failed node first, so the tasks can recover their previous state from the volumes. And we do not support dynamic provisioning, i.e. we will call CSI plugin to make pre-provisioned volumes ready for containers to use.

Non-goal

1. Non-shareable volumes.
2. Dynamic provisioning.
3. Docker containerizer support.
4. Provide some agent operator APIs for adding/updating/removing supported CSI plugins in the runtime. (see Configuration section for details)

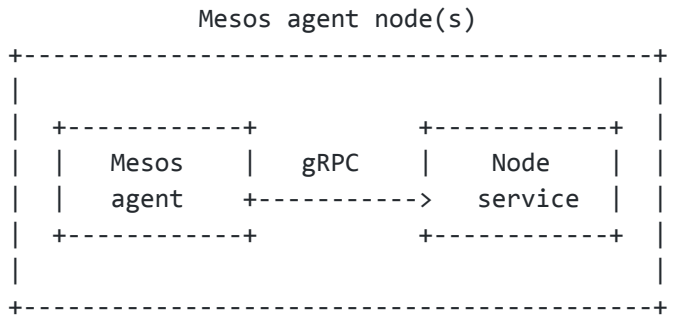
JIRA ticket

[MESOS-10141]

Design

Architecture

Usually a CSI plugin consists of two services: controller service and node service, the former can be run anywhere and the latter must be run on the node where a provisioned volume will be published for a container to use. Since in the MVP we only support shareable volumes and do not support dynamic provisioning, we will only call the node service but not the controller service.



API

A new volume type `CSI` will be added into the existing `Volume` protobuf message:

```
message ContainerInfo {
  …
  repeated Volume volumes = 2;
}

message Volume {
  message Source {
    enum Type {
      DOCKER_VOLUME = 1;
      HOST_PATH = 4;
      SANDBOX_PATH = 2;
      SECRET = 3;
```

```
  CSI_VOLUME = 5;
}
…

message CSIVolume {
  required string plugin_name = 1;

  message VolumeCapability {
    message BlockVolume {
    }

    message MountVolume {
      optional string fs_type = 1;
      repeated string mount_flags = 2;
    }

    message AccessMode {
      enum Mode {
        UNKNOWN = 0;
        SINGLE_NODE_WRITER = 1;
        SINGLE_NODE_READER_ONLY = 2;
        MULTI_NODE_READER_ONLY = 3;
        MULTI_NODE_SINGLE_WRITER = 4;
        MULTI_NODE_MULTI_WRITER = 5;
      }

      required Mode mode = 1;
    }

    oneof access_type {
      BlockVolume block = 1;
      MountVolume mount = 2;
    }

    required AccessMode access_mode = 3;
  }

  message StaticProvisioning {
    required string volume_id = 1;
    required VolumeCapability volume_capability = 2;
    optional bool readonly = 3;
    optional Secret node_stage_secret = 4;
    optional Secret node_publish_secret = 5;
    map<string, string> volume_context = 6;
  }

  optional StaticProvisioning static_provisioning = 2;
```

```
  }

  optional Type type = 1;
  ...
  optional CSIVolume csi_volume = 6;
}
```

The fields in the newly introduced `CSI.StaticProvisioning` protobuf message are merged from two protobuf messages `NodeStageVolumeRequest` and `NodePublishVolumeRequest` defined in CSI spec, the `staging_target_path` and `target_path` fields are not included in the `CSI.StaticProvisioning` protobuf message since they will be internally determined by Mesos when calling CSI plugins. In future when we support dynamic provisioning, we can introduce another protobuf message `CSI.DynamicProvisioning` whose fields should be from the `CreateVolumeRequest` protobuf message defined in CSI spec.

We will make three changes to the existing `CSIPluginInfo` protobuf message:

```
message CSIPluginInfo {
  required string type = 1;
  optional string name = 2;
  repeated CSIPluginContainerInfo containers = 3;
  optional string endpoint = 4;
  optional string target_path_root = 5;
}
```

1. Change the type of the existing field `name` from required to optional. The reason that we introduced this field is CSI does not have the concept of "volume groups" which is part of LVM, so we need a field to specify the volume group managed by our LVM CSI plugin. So this is actually a field specific for the LVM CSI plugin and other 3rd party CSI plugins do not have this concept, we'd better to make it optional.
2. Introduce a new field `endpoint`. This is for the unmanaged CSI plugins (i.e. the plugin not launched by Mesos, usually deployed by operator beforehand), to call such plugin's API, we need to know its endpoint which is usually a path to a Unix domain socket on the agent host. For managed CSI plugins, this field will be ignored, instead the field `containers` will be used to launch the plugin as a standalone container.
3. Introduce a new field `target_path_root`. This is the root directory of the CSI plugin's all the volume's target paths, i.e. each volume will be published by the CSI plugin at a sub-directory under this path. Operator needs to specify this field according to the configuration of the CSI plugin, e.g. for Portworx deployed on DC/OS, this field can only be one of `/var/lib/kubelet`, `/var/lib/mesos` and `/var/lib/osd`.

Configuration

A new agent flag `--csi_plugin_config_dir` will be added to allow the operators to configure the supported CSI plugins (both managed and unmanaged plugins). The flag points to a directory on the agent node. During startup, the `volume/csi` isolator (see next section for details) will scan the directory and load json files modeled using `CSIPluginInfo` protobuf message, such `CSIPluginInfo` will be used to either launch managed CSI plugins as standalone containers or get endpoint for unmanaged CSI plugins.

In future, we may provide some agent operator APIs for adding/updating/removing supported CSI plugins in the runtime.

`volume/csi` isolator

We will add a new isolator `volume/csi` for publishing pre-provisioned CSI volumes for UCR containers to use.

During startup, this isolator will scan the directory specified via `--csi_plugin_config_dir` and load the supported CSI plugins.

When a container is being launched, this isolator's `prepare` method will be called by UCR to handle the CSI volumes (if any) in the container's `ContainerInfo`. For each CSI volume, if its CSI plugin (identified by `Volume.csi_volume.plugin_name`) is not one of the supported CSI plugins, we will just return a failure with an error message like `The xxx CSI plugin is not supported`, and then the container will fail to launch and a `TASK_FAILED` status update will be sent to framework. If the CSI plugin is one of the supported CSI plugins, then:

1. If it is a managed plugin, we will call the existing CSI service manager to launch a standalone container to run the CSI plugin and get its endpoint. Please note that for each managed plugin we will only run it once (the first time when a container tries to use a volume managed by it) on an agent host, and then it will handle all the containers which try to use the volumes managed by it afterward.
2. If it is an unmanaged plugin, we will assume it is already deployed by the operator and get its endpoint directly from the related `CSIPluginInfo`. We could consider refactoring the CSI service manager by making it support unmanaged plugins and make it's `getServiceEndpoint` method can also return unmanaged plugins's endpoint.

With the endpoint, we can then call CSI plugin's `NodeStageVolume` (if the CSI plugin has the `STAGE_UNSTAGE_VOLUME` capability) and `NodePublishVolume` APIs to publish the volume to a target path on the agent host.

Finally the `prepare` method will return a `ContainerLaunchInfo` object with some `ContainerMountInfo` objects added, each `ContainerMountInfo` object is for bind mounting a CSI volume from its target path to the container path in the container's mount namespace.

There is an existing VolumeManager which is like a wrapper for various CSI gRPC calls, we could consider leveraging it to call CSI plugins rather than making raw CSI gRPC calls in `volume/csi` isolator. But there is a problem, the lifecycle of the volumes managed by VolumeManager starts from the `createVolume` CSI call, but what we plan to support in MVP is pre-provisioned volumes, so we may need to refactor VolumeManager by making it support pre-provisioned volumes.

When a container is being destroyed, this isolator's `cleanup` method will be called by UCR and it will call CSI plugin's `NodeUnpublishVolume` and `NodeUnstageVolume` (if the CSI plugin has the `STAGE_UNSTAGE_VOLUME` capability) APIs to unpublish the CSI volume if there are not any other containers using the same volume (so we may need to maintain a reference count for each CSI volume in the `volume/csi` isolator).