Overview

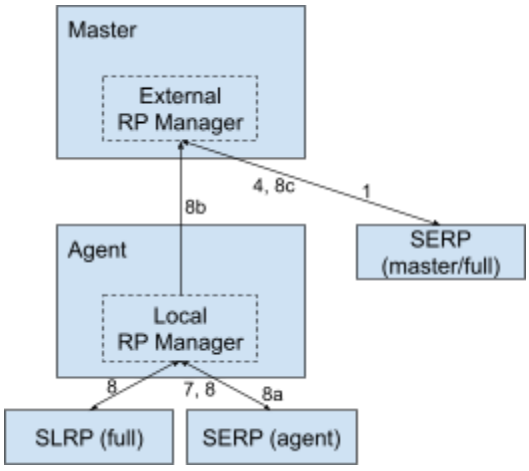
Mesos manages local storage resources through *Storage Local Resource Providers* (SLRP). SLRP should support the following MVP functions:

- Registering to the RP manager (P0)
- Reporting available disk resources through a CSI controller plugin. (P0)
- Processing resource converting operations (`CREATE_BLOCK`, `CREATE_VOLUME`, `DESTROY_BLOCK`, `DESTROY_VOLUME`) issued by frameworks to convert RAW disk resources to mount or block volumes through a CSI controller plugin (P0)
- Publish/unpublish a disk resource through CSI controller/node plugins for a task (P0)
- Module support for storage profiles (P1)
- Tracking and checkpointing resources and reservations (P1)

Design for Storage Resources Providers

A resource provider (RP) should be able to run in the following two modes: 1. a *master* mode to serve the actual resources; 2 an *agent* mode to publish the served resources to the agent it is running on; 3. a *full* mode that combines both master and agent modes. Any local resource provider, including SLRP, always runs in full mode. External resource providers such as SERP, could be deployed in two ways: 1. running in master mode on a master mode and agent mode on all agents using the resource; 2. running in full mode on an agent and in agent mode on all other agents using the resource.

The workflow is illustrated as follows:



1. S*RP master subscribes to RP Manager
2. Master sends an offers to Framework
3. Framework accepts the offer with a resource conversion request.
4. Master forwards the resource conversion request to S*RP master

5. Master schedules tasks that require the converted resource on Agent
6. Agent asks Local RP Manager to publish resources
7. Local RP Manager starts S*RP agent idempotently. For SLRP, the agent is the same as the master and thus this should be a no-op.
8. Local RP Manager asks S*RP Agent to publish the resources. SERP agent may:
    a. Sends a resource publishing request to Local RP Manager
    b. Local RP Manager forwards the request to External RP Manager
    c. External RP Manager forwards the request to SERP master
9. Agent launches tasks with the resources
10. Local RP Manager asks S*RP agent to unpublish the resources. SERP agent may talk to SERP master, similar to step 8.

An alternative to step 8 for SERP is that we could make a connection from the SERP agent directly to the external RPM or the SERP host. Further evaluation is required and is beyond the scope of this document.

Resource Provider API Extension

We introduce the following extension in the resource provider API:

```
// Protocol buffer for custom operations that a resource provider agent
// can ask the host to perform. This is transparent to Mesos and the
// resource provider host and agents should agree on a common operation
// code scheme.
message HostOperation {
  required ResourceProviderID provider_id = 1;
  required string code = 2;
   map<string, string>parameters = 3;
}

message Event {
  enum Type {
    UNKNOWN = 0;

    SUBSCRIBED = 1;
    OPERATION = 2;
    RELOAD = 3;          // See `Reload` below.
    PUBLISH = 4;         // See `Publish` below.
    UNPUBLISH = 5;       // See `Unpublish` below.
    HOST_OPERATION = 6; // See `HostOperation` above.
  }

  // Received when the master wants to reload the resource provider.
  message Reload {
    // If `config` is provided, then the resource provider should apply
    // the new configuration specified in `config`.
    optional ResourceProviderInfo config = 2;
```

```
  }

  // Received when the master wants to publish the specified resources
  // on this agent.
  message Publish {
    repeated Resource resources = 1;
  }

  // Received when the master wants to unpublish the specified resources
  // on this agent.
  message Unpublish {
    repeated Resource resources = 1;
  }

  optional Type type = 1;
  optional Subscribed subscribed = 2;
  optional Operation operation = 3;
  optional Reload reload = 4;
  optional Publish publish = 5;
  optional Unpublish unpublish = 6;
  optional HostOperation host_operation = 7;
}

message Call {
  enum Type {
    UNKNOWN = 0;

    SUBSCRIBE = 1;
    UPDATE = 2;
    POST = 3;           // See `Post` below.
    HOST_OPERATION = 4; // See `HostOperation` above.
  }

  message Subscribe {
    enum Mode {
      // This must be the first enum value in this list, to
      // ensure that if `state` is not set, the default value
      // is UNKNOWN. This enables enum values to be added
      // in a backward-compatible way. See MESOS-4997.
      UNKNOWN = 0;

      MASTER = 1;
      AGENT = 2;
    }

    required ResourceProviderInfo resource_provider_info = 1;
    repeated Resource resources = 2;
    repeated Mode mode = 3;
  }

  // Notify the master about the latest resource information.
  message Post {
    repeated Resource resources = 1;
```

```
  }

  optional ResourceProviderID resource_provider_id = 1;

  optional Type type = 2;
  optional Subscribe subscribe = 3;
  optional Update update = 4;
  optional Post post = 5;
  optional HostOperation host_operation = 6;
}
```

Configuring SLRP

Protobuf definition for a CSI plugin :

```
message CSIPluginInfo {
  required CommandInfo command = 1;
  repeated Resource resources = 2;
  optional ContainerInfo container = 3;
}
```

The `ResourceProviderInfo` protobuf would be extended as follows to support local/external storage resources:

```
message ResourceProviderInfo {
  message StorageInfo {

    map<string, CSIPluginInfo> csi_plugins = 1;
    required string controller_plugin = 2;
    required string node_plugin = 3;
  }

  optional ResourceProviderID id = 1;
  repeated Attribute attributes = 2;
  required string type = 3;
  required string name = 4;
  optional StorageInfo storage = 5;
}
```

An example SLRP configuration with split-component CSI plugins:

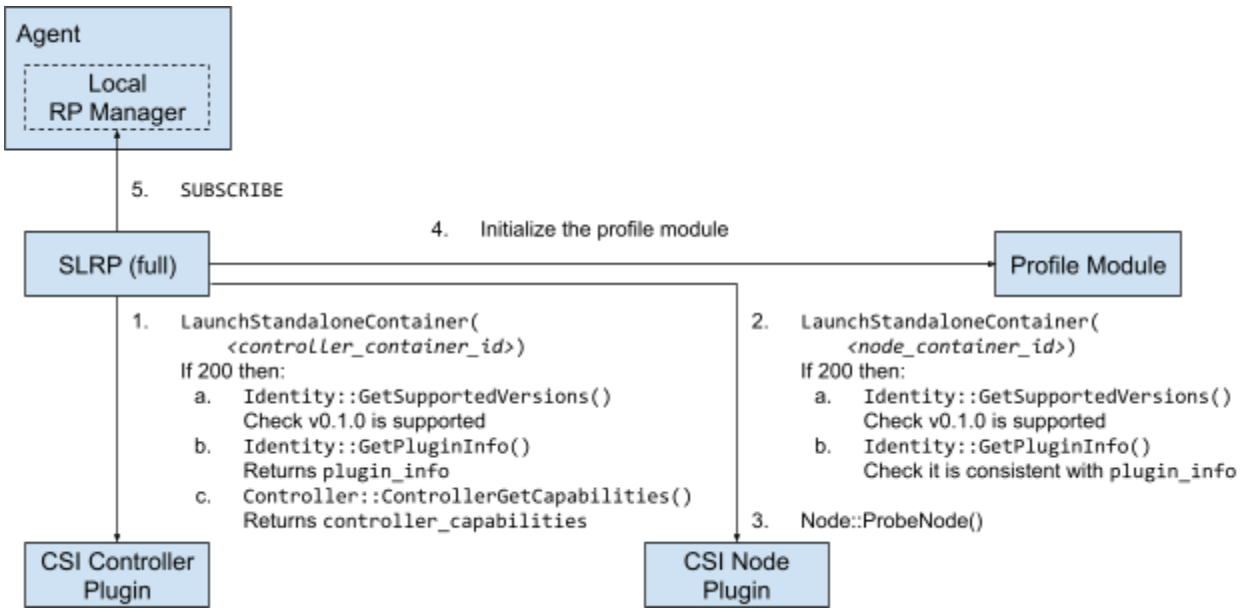
```
{
  "type": "org.apache.mesos.rp.local.storage",
  "name": "local-volume",
  "storage": {
```

```
    "csi_plugins": {
      "plugin_1": {
        "command": {...},
        "resources": [...],
        "container": {...}
      },
      "plugin_2": {
        "command": {...},
        "resources": [...],
        "container": {...}
      }
    },
    "controller_plugin_name": "plugin_1",
    "node_plugin_name": "plugin_2"
  }
}
```

Unified CSI plugins can also be supported by only providing one single plugin definition and set both `controller_plugin_id` and `node_plugin_id` to the name of the plugin.

SLRP Startup

The normal startup workflow of SLRP is shown in the following diagram. The recovery mechanism is not described here.



Launching CSI Plugins

SLRP runs CSI plugins in standalone containers. The container ID of each plugin is composed as follows:

```
mesos-rp-local-storage-<rp_name>-<plugin_name>
```

where *<rp_name>* and *<plugin_name>* are *percent-encoded* to escape '/', '.', and '-' characters. For example, the ContainerIDs of the plugins in the above configuration are

```
mesos-rp-local-storage-local%2Dvolume-plugin_1
mesos-rp-local-storage-local%2Dvolume-plugin_2
```

In the case of a unified plugin, these two IDs would be identical and hence SLRP would only start a single container because of the idempotency of `LaunchStandaloneContainer`.

Each container is launched based on the corresponding plugin definition in `csi_plugins`, with the `CSI_ENDPOINT` environment variable set to:

```
unix:///tmp/mesos-rp-local-storage-<rp_name>/<plugin_name>.sock
```

where *<rp_name>* is the path to the sandbox directory in the standalone container. The path is the same between the host filesystem and the container filesystem.

Subscribing to Resource Provider Manager

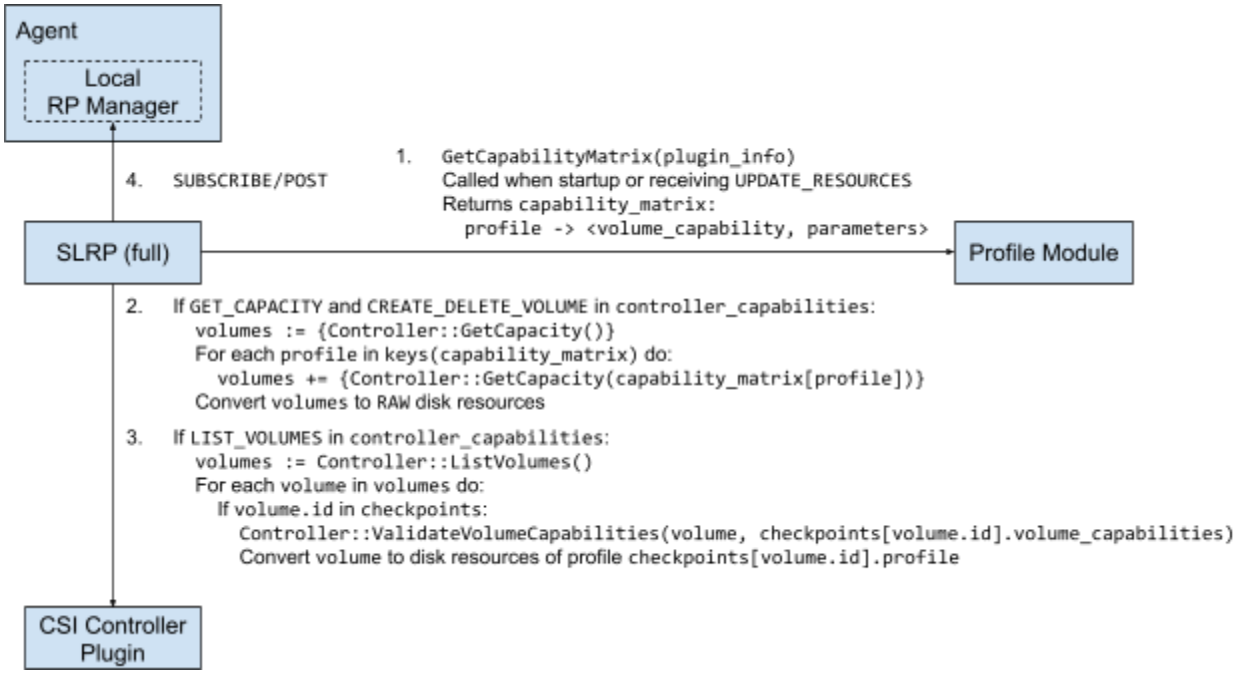
SLRP should always subscribe to the local RP manager in host mode. Prior to the subscription, it could collect the information about the profiled storage resources (described in the next section) and report that to the local RP manager along with the subscription. The local RP manager would then report the resources to the master.

Recovery Logic

If any of Step 1, 2, and 3 fails, SLRP would subscribe to the local RP manager with no resource. It can use the `POST` API call to report the RP manager once the plugins are up and running.

Resource Reporting Cycle

The workflow of collecting and reporting the storage resources is shown in the following diagram.



The above flow would be triggered before SLRP subscribes to the local RP manager, such that SLRP could report the resources along the subscription. We require both `GET_CAPACITY` and `CREATE_DELETE_VOLUME` to report unprovisioned space since we need the later to provision a volume to use the space.

Recovery Logic

If either Step 2 or 3 fails, SLRP will report that no resource is available with a `POST` API call.

Updating Resources and Profiles

We can use the `RELOAD` and `POST` API calls to dynamically add and remove local storage resources without restarting SLRP. The following list some scenarios the RP manager might want to issue a `RELOAD` event to SLRP:

- Storage profile changes (activation/deactivation)
- Resource changes (e.g., EBO quota changes from the system operator)

The mechanism to trigger a `RELOAD` event is beyond the scope of this document. Potentially, it might be triggered by either an operator API or a framework API. When SLRP receives a `RELOAD` event, it should perform the resource reporting flow described earlier after necessary updates, and then issue a `POST` call to update the resource information to the RP manager.

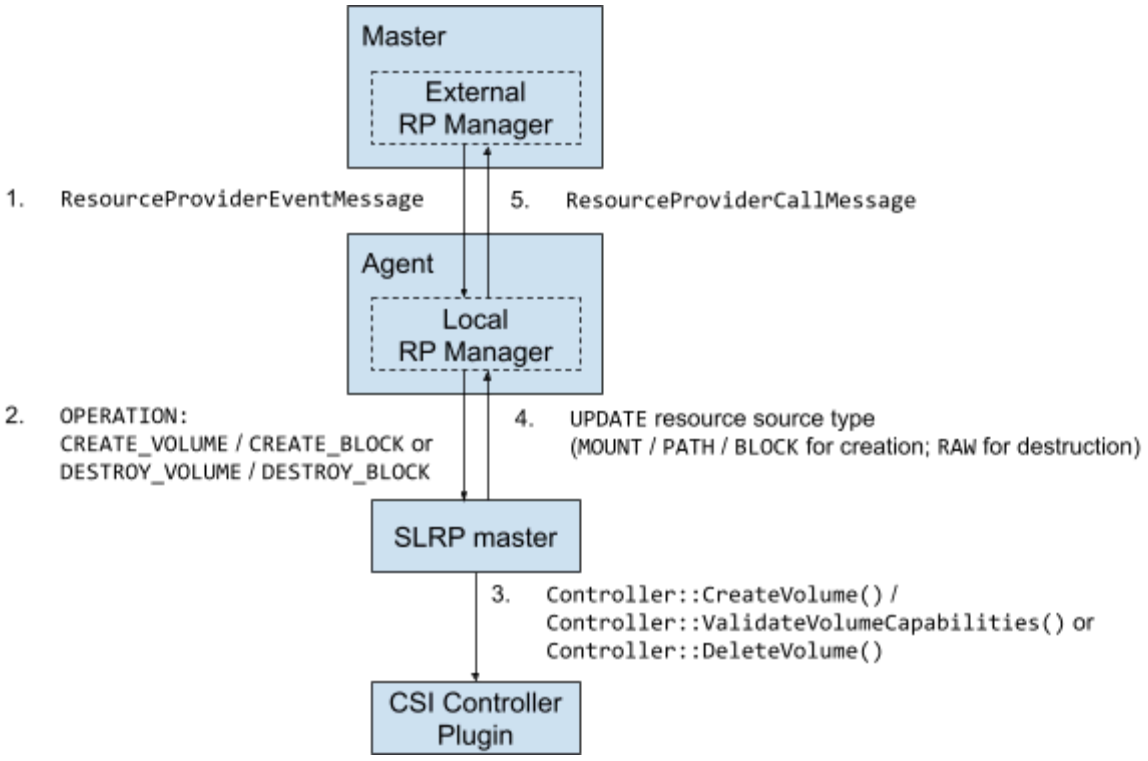
SLRP can also actively issue a `POST` call when there is any resource change, such as when a volume becomes unreachable.

Resource Conversion Cycle

A system operator may ask SLRP to create/destroy a `BLOCK` or `VOLUME` disk resource from a `RAW` disk resource. The following changes to the the `DiskInfo` protobuf are introduced, instead of those those proposed in the [storage support design doc]:

```
/**
 * Describes a CSI volume. The values of its fields are opaque to Mesos.
 */
message CSIVolumeHandle {
  required string id = 1;
  map<string, string> metadata = 2;
}

message Resource {
  message DiskInfo {
    message Source {
      enum Type {
        UNKNOWN = 0;
        PATH = 1;
        MOUNT = 2;
        BLOCK = 3;
        RAW = 4;
      }

      message Path {
        optional string root = 1;
      }

      message Mount {
        optional string root = 1;
      }

      // A block device set up by the Agent adminitsrator.
      message Block {
        optional Device device = 1;
      }

      required Type type = 1;
      optional Path path = 2;
      optional Mount mount = 3;
      optional Block block = 4;

      // A handle for a CSI plugin to identify the underlying volume.
      optional CSIVolumeHandle csi_handle = 2;

      // An identifier for the storage profile the disk belongs to.
      optional Labels profile = 5;
    }
  }
}
```

The following diagram illustrates the flow of creating/destroying dick resources:



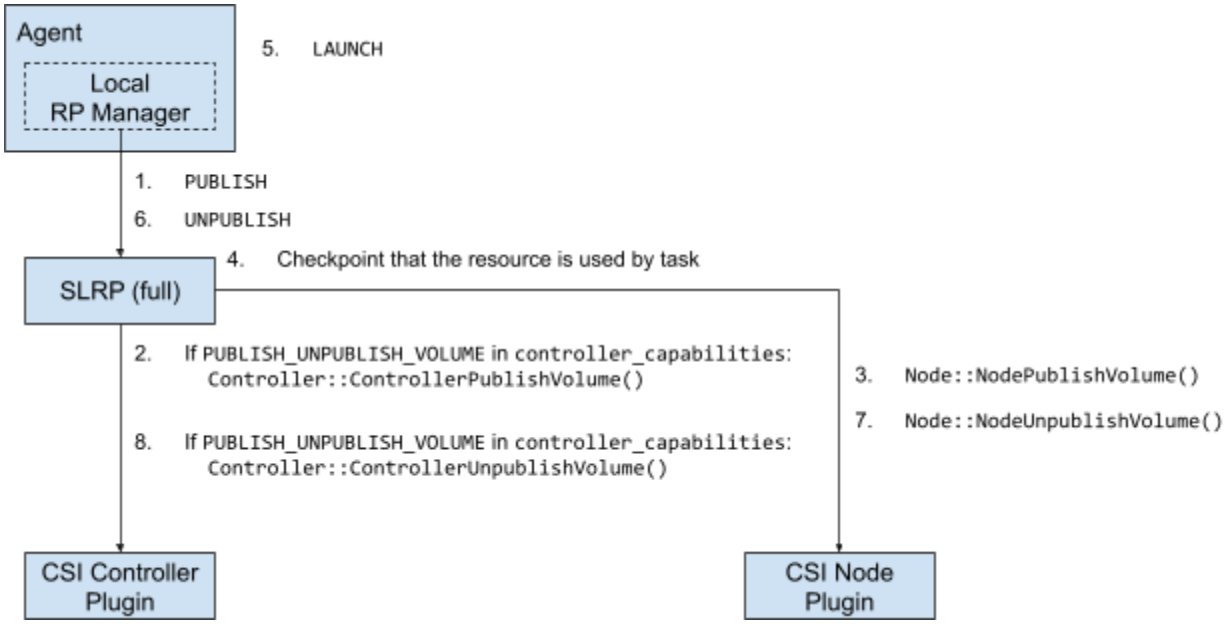
For disk resource creation, Step 3 depends on if the resource to convert is an unprovisioned space or a provisioned volume. If it is unprovisioned, SLRP calls `CreateVolume` to provision it; otherwise it calls `ValidateVolumeCapabilities` to make sure that the volume can be published with the specified type.

Recovery Logic

If Step 3 fails, SLRP should make an `UPDATE` call with a `FAILED` state, and the agent will forward this failure to the master. Eventually the master would tell the framework that the offer operation is failed.

Resource Consuming Cycle

Upon receiving a task, the local RP manager asks SLRP to publish the specified resources on the node. The specified resources are unpublished once the task is finished. The following diagram illustrates the workflow of managing the storage resource used by a task:



The local RP manager sends PUBLISH and UNPUBLISH that we introduced in the resource provider API earlier to ask SLRP to publish and unpublish the storage resources required by a task.

Recovery Logic

If Step 2 or 3 fails, the task should be considered in the TASK_DROPPED state. If Step 7 or 8 fails, then we need a way to tell the master that the task is finished but the resource is not returned.

Discussion: SERP

For external storage support in the future, we will introduce an *Storage External Resource Provider* (SERP). SERP runs in host mode on a master node or could be deployed by Marathon, and should be an HA service. SERP agents can be initialized by the local RP manager when it receives tasks requiring external storage resources before Step 1. However, we will need to pass in the whole `ResourceProviderInfo` instead of just a `ResourceProdiverID` in the `Resource` protobuf to initialize the SERP agent.

To support centralized CSI plugins, the SERP agent might need to ask the host to perform certain `HOST_OPERATION` (such as CSI `ControllerPublishVolume` or `ControllerUnpublishVolume`) for `PUBLISH` and `UNPUBLISH` in Step 1 and 6.

Interaction with CSI Plugins

Plugin Lifecycle Management

We would introduce a daemon process class to manage the standalone containers for the CSI controller and node plugins. The daemon class should provide the following functions:

- Launch the plugin describe in SLRP Startup. The plugin is considered successfully launched only when all necessary CSI calls succeed.
- Relaunch the plugin with an updated configuration.
- Relaunch the plugin when there is no heartbeat or if any CSI call fails.
- Notify SLRP when the plugin is down and available again. SLRP should use this information to send proper `POST` API calls.
- Support CSI calls with retries.
- Terminate and wait for the termination of the standalone container asynchronously.

Operation Retry Logic

There is no way to make sure that the plugin is in a healthy state when a CSI call is made. Even if we have heartbeats or health check for standalone containers, there is still no guarantee that the plugin is ready to serve. Therefore, we would like to retry a CSI call when an error is returned, and only consider it as "failed" after several retry. The retry logic might look like:

1. If a CSI call returns an error, relaunch the plugin.
2. If the relaunch is successful, retry the CSI call. We may consider to have multiple retries.
3. Return a failure if the retry is not successful.

Related Docs

Improve Storage Support in Mesos using Resource Provider and CSI
Container Storage Interface (CSI) Specification
Standalone Container Daemons