Supporting Stateful Services in Mesos

Motivation and goal

Our ultimate goal is to provide support for running stateful services (e.g., distributed filesystem or database) on Mesos. One of the key features is to provide a way for frameworks to have guaranteed access to their lost state even after task failover (or slave restart). If a stateful service task dies on a Mesos slave, Mesos currently makes no guarantees that any of the freed (or even future available) resources from that slave will be reoffered to the same framework so that it can launch a task there to recover its lost state. In fact, Mesos will eventually garbage-collect the task's sandbox, so "persistent" data on disk could be lost as well, even if the framework were eventually able to run on that slave again.

Primitives to build

We will need to build a few new primitives in Mesos in order to achieve the above goal. In this section, I will list these primitives. Some of the primitives are required and some of them are recommended. When I say recommended, it means there exist some workarounds for frameworks but with some limitations.

P1: Persistent disk resource

Mesos needs to provide a way to allow tasks to write persistent data which won't be garbage collected. For example, a task can write its persistent data to some predefined directory. When this task finishes, the framework can launch a new task which is able to access the persistent data written by the previous task which Mesos would have usually garbage-collected.

One way to achieve that is to provide a new type of disk resources which are persistent. We call it *persistent disk resource*. When a framework launches a task using persistent disk resources, the data the task writes will be persisted. When the framework launches a new task using the same persistent disk resource (after the previous task finishes), the new task will be able to access the data written by the previous task.

The persistent disk resource should be able to survive slave reboot or slave info/id change.

As a workaround, a framework might choose to write persistent data to an out-of-band location (e.g., /var/lib/cassandra). However, nothing prevents two frameworks competing on the same out-of-band location, causing potential undefined behaviors.

P2: Lazy/Dynamic Resource reservation

Mesos needs to provide a way to allow frameworks to reserve resources (e.g., cpu, memory, etc.) for their stateful tasks upon launching so that the stateful tasks can always be re-launched on the same slave that contains their persisted state without worrying about those resources being allocated to other frameworks. For example, the HDFS framework may want to reserve 1cpu, 2G memory on a slave where it is running a task (and storing state) so that it will always be able to re-launch a data node task on that slave, even if the original task dies.

In fact, the reservation problem can be addressed by the existing role-based *static resource reservation*. However, that requires operators to set up role based reservation ahead of time (at slave start time) on each slave for each framework/role that wants to support stateful services, which might be very tedious. To mitigate that, we have a couple of choices. We can leverage the master reservation described in MESOS-1791, or add a new primitive which allows frameworks to dynamically reserve resources while launching tasks.

The reservation should be able to survive slave reboot or slave info/id change. This is very important because a stateful service will still want to get on to the box where its persistent data are written even after slave reboots.

P3: Disk isolation

This is a more like a general topic than supporting stateful service. We need to provide isolation for disks which we do not provide currently:

Disk IO isolation (MESOS-350)
Filesystem isolation (MESOS-1589)
Disk quota isolation (MESOS-1588)
Support disk spindles (MESOS-191)

P4: Dynamic slave attributes, modifiable by frameworks

Imagine a framework whose persistent state is not covered by the above persistent disk resources (e.g. NVRAM or another unpredicted resource type). A framework using such persistent state would like to be able to annotate the slave hosting that state to indicate a) the presence of the state on that slave, b) what kind of state is hosted there, and/or c) where and how to access that state. Building on the work of MESOS-1739, a framework could conceivably add new attributes to a slave (perhaps only readable by the same

framework/role) dynamically, at task launch time. This mechanism could be used by frameworks to store per-slave metadata about persistent state.

The current workaround would be for each framework to track and persist this metadata on its own, and it would be difficult to share such metadata with other frameworks.

Primitive Priority

I believe the priority should be: P1 -> P2 -> P3 -> P4.

Overview

```
message Resource {
  required string name = 1;
  required Value.Type type = 2;
  optional Value.Scalar scalar = 3;
  optional Value.Ranges ranges = 4;
  optional Value.Set set = 5;
  optional string role = 6 [default = "*"];
  optional string reserved_role = 7;

  message DiskInfo {
    required string id = 1;
    optional Volume volume = 2;
  }

  optional DiskInfo disk = 8;
}

message Volume {
  // Absolute path pointing to a directory or file in the container.
  required string container_path = 1;

  // Absolute path pointing to a directory or file on the host.
  optional string host_path = 2;

  enum Mode {
    RW = 1; // read-write.
    RO = 2; // read-only.
  }

  required Mode mode = 3;
}
```

The 'reserved_role' is used to for dynamic resource reservation while 'role' is used for static resource reservation.

When launching a task, a framework can set 'reserved_role' to express its intention to dynamically reserve that resource (if it hasn't been set). The ‘reserved_role’ should be the role the framework registers with Mesos via its FrameworkInfo. In the future, a framework will be allowed to register with multiple roles (MESOS-1763).

The 'reserved_role' needs to be compatible with the static 'role'. For example, it's OK to have role = '*' and reserved_role = 'aurora'. It's not ok to have role = 'aurora' and reserved_role = 'hdfs'.

When the task finishes, the resources with 'reserved_role' set will be re-offered to the corresponding frameworks (in that role). When a framework receives a resource with 'reserved_role' set, it knows that this is a resource that is dynamically reserved. If it no longer needs that reservation, it can release it by using a new API called 'release' to release the dynamically reserved resources. However, it cannot release a statically reserved resource!

To support persistent disk resources, we add an optional 'DiskInfo' in Resource. It's only meaningful if resource.name = 'disk' (towards first class resource!). The id is the handle we mentioned in the previous design doc which is used to identify the persistent directory. The same id will be returned to the framework through offers to allow accesses to the persistent data.

We also introduce an optional Volume to support pre-existing persistent directory (e.g., /var/lib/cassandra) as one can explicitly specify the ‘host_path’ in Volume.