Design Doc - Self-healing

THIS IS A WORK IN PROGRESS.

Motivation

One of the problems facing operators of a Mesos deployment is maintenance of the hosts that the service runs on. As the size of the fleet grows, the importance of getting the service to be capable of being run in a truly, "lights-out" mode becomes even more pronounced. This is largely because the cost of maintaining the service operationally can grow in equal or greater proportion to the size of the nodes being added to it.

What's more, it's believed that by taking node health into consideration the scheduling efficiencies.

In order to solve these problems, this project proposes the introduction of self-healing infrastructure to the master and slave nodes. This infrastructure would allow for the carefully coordinated execution of a pluggable repair commands which will be used to attempt to, "repair" node issues through a common set of actions such as: node rebooting, re-imaging and finally node isolation.

Design

Notification mechanism

The first requirement in designing a system which performs automated repair is to define a mechanism by which signal can feed into the service. This signal can then be processed and used to determine what repairs make sense to execute based not only on the problem (the signal) but also on which repairs have been previously attempted.

In order to provide such a mechanism it's proposed that we introduce a new component to the mesos-master process a, "Repair Coordinator" which can be used to process the signal we provide it. In order to get this signal to this new component we have several options. These are as follows:

1) **HTTP** - Extend the existing Master::Http implementation to accept POST requests at a new, "/repair" url. This would allow for easy integration by a variety of monitoring infrastructures and for easy interaction, should it be desired, from other tools like curl.

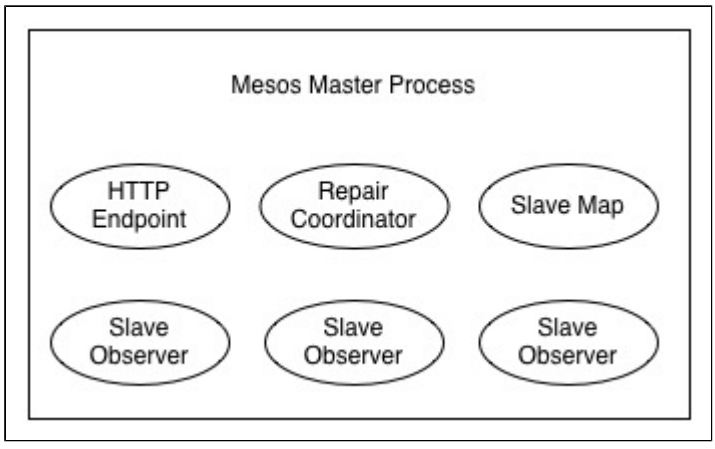
2) **Protobuf** - Introduce new protobuf messages which the master would process as repair reports. The obvious disadvantage here being that it sending messages to the master in this manner may make it more difficult to integrate with existing monitoring infrastructure that users of Mesos may already have.

The current thinking is that we will lean towards option #1 since it makes the reuse of existing infrastructure that Mesos users may already have. This is the certainly the case at Twitter and so it's imagined others will find themselves in similar positions.

Once we have the ability to get signal we will need to process that. As alluded to earlier in the doc we will introduce a new component to process the signal that comes. It will be the responsibility of this component to ensure the following:

- That no more then the user configured number of repairs are executed at any one time. This is to ensure that we don't attempt mass repair of the Mesos environment which may result in service outages.
- That we choose an appropriate repair to execute. The coordinator should remember the set of repairs which have been previously attempted and should choose the next repair in a kind of escalating set of repair actions.
- In the case where no repair is able to repair a host the coordinator should isolate the slave node from future use by the registered frameworks to avoid task execution errors.

Figure 1 provides a simplified illustration of the augmented mesos-master process with the new, "Repair Coordinator".



Once we have the ability to receive signal on the mesos master for problems that are taking place we need a mechanism which will actually send it. This design makes the assumption that this signal could come from one of the following mechanisms.

1) **In-house monitoring** - In this configuration it's assumed that a company has developed its own monitoring tech and would like to continue to reuse it to feed input to Mesos.

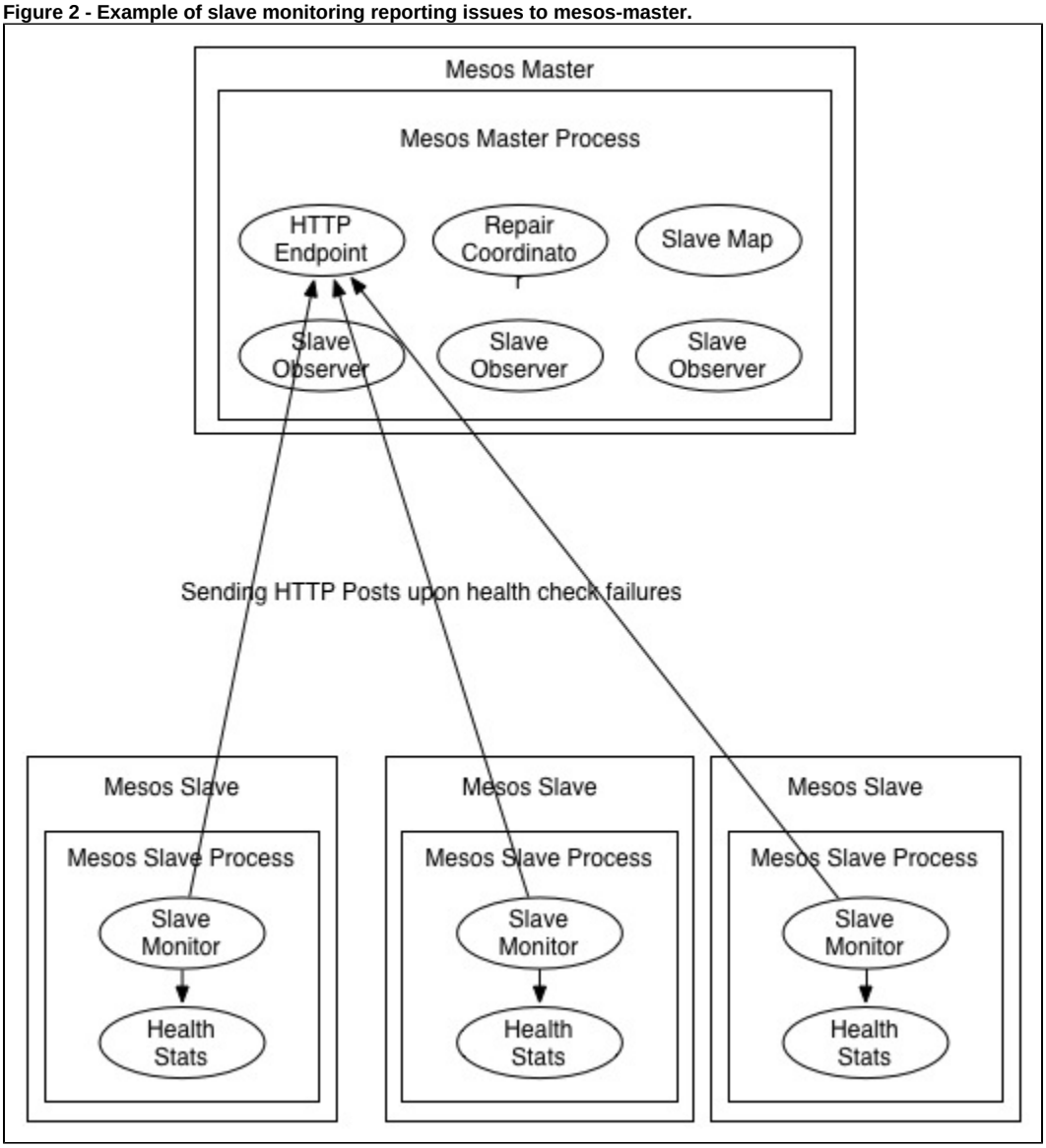
2) **Mesos slave monitoring** - In this configuration a new monitoring agent will be added to the slaves. This will communicate back to the master when problems are found on the slaves themselves thus acting as the trigger for repairs.

For the in-house monitoring based solution it's assumed that the monitoring service will have pre-defined thresholds for safe operating conditions. Then, when these thresholds are crossed, that the they are capable of sending a HTTP POST for length of time that the slaves remain outside of the safe operating thresholds.

In the Mesos slave monitoring solution (shown in Figure 2), the slave processes will use a monitoring thread which will rely upon a configuration file that's deployed along with the slave. This configuration file will specify the following:

1. The set of operational statistics to collect
2. The envelope around each statistic that is deemed to be the safe operating threshold.
3. The collection interval in minutes. NOTE: From prior experience, sub-minute collection intervals rarely tend to be beneficial enough to merit the increased cost of collection.

Figure 2 - Example of slave monitoring reporting issues to mesos-master.



Upon detecting a violation, the slave monitor will report this case to the mesos master on the newly extended HTTP endpoint which was described earlier. This notification will result in a call to the RepairCoordinator to process the notification.

Slave Repair

Before the coordinator can issue a repair it first must be made aware of a couple of things.

1) The total set of repairs that are available for the coordinator to choose from.

2) The order in which repairs should be attempted. As an example, it's likely best to try restarting a host before attempting to re-image it given the cost in time of the repair.

3) The scope that the repair should take place at. Some repairs are executed on the slave itself (such as rebooting) others may need to be executed by the master if the slave is unresponsive or incapable of receiving the repair request.

4) The number of concurrent repairs which are allowed. This is to ensure that we don't have a rush of repairs that result in a service outage.

Given this, this design proposes that we add several new flags to the the master and slave processes. These are as follows:

- -repair_set=<repair_id>=<path_to_cmd>,<repair_id>=<path_to_cmd>
- -repair_order=<repair_id>:<scope>,<repair_id>:<scope>
- -num_allowed_repairs=<integer>

The first flag specified will be used to make an association between a specific repair_id (which must be unique) to a specific command to execute in order to issue the repair. The second flag establishes both the order of the repairs but also the scope at which the repair should be executed. If a repair_id is specified but cannot be found in the repair_set then a error will be issued. The scope must be equal to either, "master" or "slave". Any other value specified will result in an error at startup time.
The final flag is fairly self explanatory. In short, it will be used to control the number of concurrent repairs that will be allowed to take place. This will be taken into consideration by the RepairCoordinator before issuing repairs to slaves.

Implementation Plan

It's proposed that this project be carried out in several stages.

1. **Listening** - - In this segment we will focus on delivering the HTTP endpoint extensions necessary with Mesos Master. We will also introduce the RepairCoordinator as well but this will not actually do any repairs.
2. **Reacting** - In this segment we will introduce the flags at the master and slave level. We will also introduce the RepairExecutor component though it will still just to continue to log what it would have done.
3. **Executing** - In this segment we will actually execute the repairs at the slave and master level. The service will honor the escalation order and will have all of the stats for monitoring as described here.