on Performance Metrics in Apache Mesos

Backing SLA and QoS in Mesos - Version 0.0.1

Introduction

In the context of oversubscription, but useful as a general feature; allowing frameworks to report their Service Level Indicators (SLIs) and Service Level Objectives (SLOs) (to which degree, if violated, etc) in a homogeneous way allows for better monitoring, scheduling, auto-scaling and more aggressive oversubscription strategies.

User stories

1. As a Mesos framework author, I want a way to describe how to obtain performance metrics for the tasks that my scheduler launches. As a Mesos framework author, I want mesos to provide an interface for declaring SLOs and allow SLIs to be provided for oversubscription consideration.

2. As an operator, I want a unified way to determine the performance of my cluster workloads.

3. As an operator, I want my performance metrics to represent local and global performance.

4. As an oversubscription module developer, I want to know if I am hurting my production workloads by making bad resource estimates or failing to correct them.

5. As an operator, I want metrics to be machine parsable i.e. not too free-form to have to maintain translations,

6. As a developer with an existing application that is reporting application metrics (such as statsd or collectd) to be able to report SLIs.

Terminology

Short	**Term**	**Description**	**Example**
APM	Application Metric	SLI and SLO	
SLI	Service Level Indicator	A measurement of workload performance.	95% tail request latency.

SLO	Service Level Objective	A threshold, based on an SLI.	200 milliseconds maximum for 95% tail request latency.
SLA	Service Level Agreement		“The thing that describes what happens when your SLI doesn’t meet your SLO” ~ John Wilkes
QoS	Quality of Service		

Requirements

- SLI + SLO are available from slave endpoint. For example:
    - /monitor/sli
    - /monitor/slo
- SLI + SLO are made available for estimator + QoS Controller
- SLI + SLO are modeled with protobuf in set per metric
- SLOs originate from the Scheduler

Design exploration

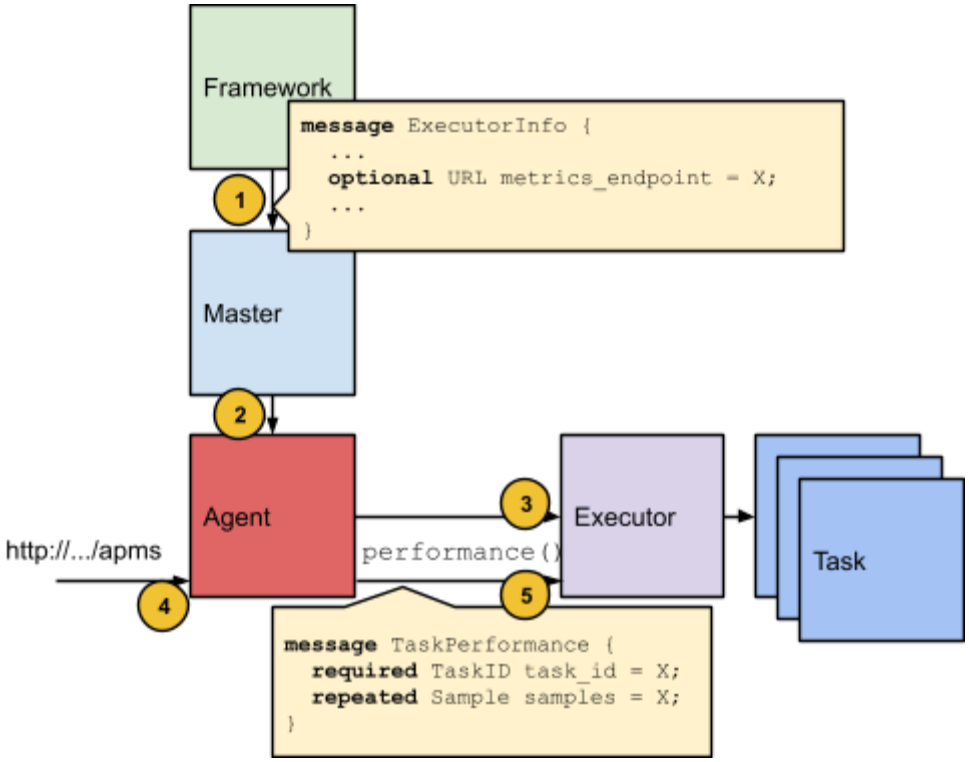
- *Option 1:* Mesos predefines APM structure (in form of protobuf message and corresponding JSON) and `TaskInfo` encodes URL to get it
    - Pros:
        - First class in Mesos (encourages framework writers to supply this info.)
        - No need for more process control in the Mesos slave.
    - Cons:
        - Templating APM URLs with framework, executor, and task IDs could be complicated (for example, http://<ip and port of agent>/stats/<task id>).
        - Burdens user with process control (need to start another service or modify their task to listen on a new endpoint.)
        - In extreme cases of resource contention, metrics may be unavailable.
        - The Mesos slave would need to manage client connections to APM endpoints.
- *Option 2:* Mesos predefines APM structure and `TaskInfo` encodes a command to fetch it.
    - Pros:
        - First class in Mesos (encourages framework writers to supply this info.)
            - Follows precedent of existing slave components (executors, health check programs, ...)

        - Potentially lighter-weight than hosting a local HTTP service.
        - Potentially easier to provide metrics for unmodified third-party applications.
        - Requires fewer changes to executor and/or task code.
    - Cons:
        - Requires more process control in the Mesos slave.
        - Depends on tools available on the slave or in the container (configuration dependent).
- *Option 3:* Out-of-band. Oversubscription modules define APM structure and listen for metric samples, e.g. on an HTTP endpoint.
    - Pros:
        - Virtually no changes to Mesos
    - Cons:
        - Not standardized across oversubscription implementations
        - Harder to interpret application specific metrics, which is one of the main drivers of this effort.
- *Option 4:* Mesos predefines APM structure and introduces a new executor callback, `performance()`.
    - Pros:
        -
    - Cons:
        - No HTTP endpoint access and harder to feed into monitoring systems
- ***Option 5:*** Combine options (1) and (4) as follows. Mesos predefines APM structure. The `TaskInfo` message gains a new optional field to describe a way to retrieve task performance (e.g. an HTTP endpoint). Introduce a new executor callback, `performance()`. The default Mesos command executor implements this new callback by fetching APM from the optionally supplied URL.
    - Pros:
        - First class in Mesos (encourages framework writers to supply this info.)
        - Provides a way for framework authors to augment current task descriptions to fetch APM without having to implement a custom executor.
        - Provides a unified way to access performance data for internal and external consumers.
    - Cons:
        - More work than other approaches, although this could be implemented in phases.

Decision: Preference is currently **Option 5**.

Prior art

Architecture



The proposed sequence is as follows:

1. A scheduler writer can choose to set a `metrics_endpoint` which indicates where to get
2.
3. The executor can be wired up to provide metrics for it's tasks. By default, the command executor will defer to the `metrics_endpoint`.
4. The agent now provides an /apms endpoint (which can be queried for all or subsets of executor/task performance metrics).
5.

```
/**
* Describes a collection of task performance metrics.
*/
message TaskPerformance {

  // p50, p90, p95, p99
  // stdev, avg, mean, min, max

  message Sample {
    required string name = 1;
    required double value = 2;
```

```
    optional double limit_upper = 3;
    optional double limit_lower = 4;
    // TODO(CD): Should this include a weight?
    // TODO(CD): Should this include severity?
  }
  required TaskID task = 1;
  repeated Sample samples = 2;
  // TODO(CD): Should this include labels?
}
```

Sidelight: Structured Metrics for Histogram Data

```
my-latency-metric: {
  count: 896270,
  max: 1847,
  mean: 437.42315175097275,
  min: 171,
  p50: 376,
  p75: 505,
  p95: 883,
  p98: 1847,
  p99: 1847,
  p999: 1847,
  stddev: 320.1211475413669
}
```

Example: JSON version of output from the codahale metrics library.

```
class Executor
{
public:
  virtual void registered(
      ExecutorDriver* driver,
      const ExecutorInfo& executorInfo,
      const FrameworkInfo& frameworkInfo,
      const SlaveInfo& slaveInfo) = 0;

  virtual void reregistered(
      ExecutorDriver* driver,
      const SlaveInfo& slaveInfo) = 0;
```

```
  virtual void disconnected(ExecutorDriver* driver) = 0;

  virtual void launchTask(
      ExecutorDriver* driver,
      const TaskInfo& task) = 0;

  virtual void killTask(
      ExecutorDriver* driver,
      const TaskID& taskId) = 0;

  virtual void frameworkMessage(
      ExecutorDriver* driver,
      const std::string& data) = 0;

  virtual void shutdown(ExecutorDriver* driver) = 0;

  virtual void error(
      ExecutorDriver* driver,
      const std::string& message) = 0;

  virtual Result<TaskPerformance> performance(
      Option<list<std::TaskID>> taskId = None()) = 0;
};
```

Recommended metrics

Metric category	Name	Description	Example	Notes
Throughput	*TODO: establish naming conventions for these categories.*	Describes how much work the task completed within a given duration.	Queries per duration *d*. Requests per duration *d*.	These metrics should not generally have upper or lower limits, except for rare cases where the request volume is constant.
Latency		Describes how quickly the task was able to respond to	95% tail request latency.	These metrics are the best kind of indicator

		input.		because they are largely independent of request volume, and they are a direct representation of typical SLO contracts.
Availability			Dropped requests over duration *d*.	
Errors			Timeouts per duration *d*. Dropped connections per duration *d*. Application exceptions per duration *d.*	These are heuristics that *could* be caused by interference. It's up to QoS policy whether these cause corrections to be issued.

TODO: Include concrete example for well-known workloads.

- Memcached
- Cassandra
- MySQL
- Apache Web Server

TODO: Design and implement a proof-of-concept QoS controller that uses APM data.

Open Questions