tapes Documentation

Release 0.0.dev20150321181547

Emils Solmanis

March 21, 2015

Contents

Contents:

CHAPTER 1

Quick start

1.1 Dependencies

The libraries dependencies are kept to a minimum to avoid the overhead for features you choose not to use, e.g.,

- tornado is not formally a dependency, but is obviously needed should you choose to use the tornado features
- pyzmq is not formally a dependency, but is needed should you choose to use the distributed reporting features

1.2 Single process

If you just need to get off the ground and flying, there's convenience methods in the `tapes` module, so you can do:

```
timer = tapes.timer('my.timer')

@app.route('/widgets')
def moo():
    with timer.time():
        return 'stuff'
```

This will simply use a global *`tapes.registry.Registry`* object instance under the hood.

You can also explicitly create and maintain a registry (or several, if needed) by just creating some:

```
registry0 = Registry()
registry1 = Registry()

timer0 = registry0.timer('some.name')
timer1 = registry1.timer('some.name')
```

Even though the timers were created with the same name, they're maintained in different registries, they're separate instances.

Repeated calls on the same registry with the same name will return the existing instance:

```
timer0 = tapes.timer('some.timer')
timer1 = tapes.timer('some.timer')
# timer0 and timer1 are actually the same instance
```

1.3 Utilities

It's common that you need to add the same metrics to many similar subclasses but have them all named differently based on the class name. Because you would usually want to make the metric a class level attribute to avoid doing the tedious lookup in particular methods, there is a utility metaclass that adds any necessary metrics to the class.

Example:

```
class MyCommonBaseHandler(tornado.web.RequestHandler):
    __metaclass__ = metered_meta(metrics=[
        ('latency', 'my.http.endpoints.{}.latency', registry.timer),
        ('error_rate', 'my.http.endpoints.{}.error_rate', registry.meter),
    ], base=abc.ABCMeta)
```

The `base` argument wires in any existing metas, most commonly `abc.ABCMeta`. The `metrics` argument takes a list of tuples, in which - the 1st arg is the name of the class attribute - the 2nd arg is a template for the metric name. It will get rendered by calling `template.format(class_name)` - the 3rd arg is a callable taking a single argument – the fully rendered name – and returning a metric instance

The above example would add two attributes to all subclasses of `MyCommonBaseHandler` called `latency` and `error_rate`, which would be a timer and a meter, and correspond to metrics parametrized on the class name.

For more info check the full docs at *`tapes.meta.metered_meta()`*.

1.4 Reporting

For pull style reporting you need to

- call *`tapes.registry.Registry.get_stats()`*
- expose the returned dict of values somehow

This should be trivial for most web applications, e.g., for Flask:

```
@app.route('/metrics')
def metrics():
    return jsonify(tapes.get_stats())
```

or Tornado:

```
class MetricsHandler(web.RequestHandler):
    @gen.coroutine
    def get(self, *args, **kwargs):
        self.finish(tapes.get_stats())
```

For push style reporting it gets more complicated because of the GIL. There are a couple of scheduled reporters that report with configurable intervals, but they create a `Thread`. For most Python implementations this essentially means that your application would block for the time it is doing the reporting, and while it is probably fine for most scenarios, it's still something to keep in mind.

There is a scheduled reporter base class for `Tornado` as well, which is non-blocking and uses the `IOLoop` to schedule reporting.

Currently the push reporting is implemented for streams and StatsD.

The reporters are all used almost the same, i.e., by creating one and calling `start()` on it:

```
reporter = SomeScheduledReporter(
    interval=timedelta(seconds=10), registry=registry
)
reporter.start()
```

If you want a guaranteed low latency setup, you might want to look into the multi-process options below.

Otherwise, take a stroll through `tapes.reporting`

1.5 Multi process

Python web applications are generally run with several forks handling the same socket, either via *uWSGI*, *gunicorn*, or, in the case of Tornado, just natively.

This causes problems with metrics, namely

- HTTP reporting breaks. The reported metrics are per-fork and you get random forks every time you make a request
- Metrics aren't strictly commutative, so even if you do push-reporting, you end up losing some info in the combiner, e.g., StatsD
- Push-reporting generally means blocking your main application's execution due to the GIL

Hence, with Tapes you have an option to

- run a light-weight proxy registry per fork
- have an aggregating master registry that does all the reporting in a separate fork

The proxy registries communicate with the master registry via 0MQ IPC pub-sub. Because 0MQ is really, **really** fast, this ends up being **faster** than just computing the metrics locally anyway.

The obvious drawbacks are

- a separate forked process doing the aggregation and reporting
- a slight lag if the traffic volume is low due to batching in 0MQ

With that said, it's much simpler to actually use than it sounds. The reporters are the same, so the only changes are the use of an aggregator and proxy registries.

NOTE: due to the way `gauge` operates, it's unavailable in the distributed mode. If / when I figure out how to combine it, I'll add it.

Tornado example:

```
registry = DistributedRegistry()

class TimedHandler(web.RequestHandler):
    timer = registry.timer('my.timer')

    @gen.coroutine
    def get(self):
        with TimedHandler.timer.time():
            self.write('finished')

RegistryAggregator(HTTPReporter(8889)).start()

server = httpserver.HTTPServer(application)
server.bind(8888)
server.start(0)

registry.connect()

ioloop.IOLoop.current().start()
```

NOTE

- `tapes.distributed.registry.RegistryAggregator.start()` is called **before** the `fork()` call
- `tapes.distributed.registry.DistributedRegistry.connect()` is called **after** the `fork()` call

Check the API docs for more info – `tapes.distributed.registry`.

CHAPTER 2

API docs

class `tapes.registry.`**`Registry`**

Factory and storage location for all metrics stuff.

Use producer methods to create metrics. Metrics are hierarchical, the names are split on '.'.

`counter`(*name*)

Creates or gets an existing counter.

Parameters **`name`** – The name

Returns The created or existing counter for the given name

`gauge`(*name*, *producer*)

Creates or gets an existing gauge.

Parameters **`name`** – The name

Returns The created or existing gauge for the given name

`get_stats`()

Retrieves the current values of the metrics associated with this registry, formatted as a dict.

The metrics form a hierarchy, their names are split on '.'. The returned dict is an *addict*, so you can use it as either a regular dict or via attributes, e.g.,

```
>>> import tapes
>>> registry = tapes.Registry()
>>> timer = registry.timer('my.timer')
>>> stats = registry.get_stats()
>>> print(stats['my']['timer']['count'])
0
>>> print(stats.my.timer.count)
0
```

Returns The values of the metrics associated with this registry

`histogram`(*name*)

Creates or gets an existing histogram.

Parameters **`name`** – The name

Returns The created or existing histogram for the given name

`meter`(*name*)

Creates or gets an existing meter.

Parameters **`name`** – The name

Returns The created or existing meter for the given name

timer(*name*)
Creates or gets an existing timer.

Parameters **name** – The name

Returns The created or existing timer for the given name

tapes.meta.**metered_meta**(*metrics*, *base=<type 'type'>*)
Creates a metaclass that will add the specified metrics at a path parametrized on the dynamic class name.

Prime use case is for base classes if all subclasses need separate metrics and / or the metrics need to be used in base class methods, e.g., Tornado's `RequestHandler` like:

```
import tapes
import tornado
import abc

registry = tapes.Registry()

class MyCommonBaseHandler(tornado.web.RequestHandler):
    __metaclass__ = metered_meta([
        ('latency', 'my.http.endpoints.{}.latency', registry.timer)
    ], base=abc.ABCMeta)

    @tornado.gen.coroutine
    def get(self, *args, **kwargs):
        with self.latency.time():
            yield self.get_impl(*args, **kwargs)

    @abc.abstractmethod
    def get_impl(self, *args, **kwargs):
        pass


class MyImplHandler(MyCommonBaseHandler):
    @tornado.gen.coroutine
    def get_impl(self, *args, **kwargs):
        self.finish({'stuff': 'something'})


class MyOtherImplHandler(MyCommonBaseHandler):
    @tornado.gen.coroutine
    def get_impl(self, *args, **kwargs):
        self.finish({'other stuff': 'more of something'})
```

This would produce two different relevant metrics,

- `my.http.endpoints.MyImplHandler.latency`
- `my.http.endpoints.MyOtherImplHandler.latency`

and, as an unfortunate side effect of adding it in the base class, a `my.http.endpoints.MyCommonBaseHandler.latency` too.

Parameters

- **metrics** – list of (attr_name, metrics_path_template, metrics_factory)
- **base** – optional meta base if other than *type*

Returns a metaclass that populates the class with the needed metrics at paths based on the dynamic class name

class `tapes.reporting.`**`ScheduledReporter`**(*interval*, *registry=None*)
Super class for scheduled reporters. Handles scheduling via a `Thread`.

`report`()
Override in subclasses.

A Python `Thread` is used for scheduling, so whatever this ends up doing, it should be pretty fast.

class `tapes.reporting.http.`**`HTTPReporter`**(*port*, *registry=None*)
Exposes metrics via HTTP.

For web applications, you should almost certainly just use your existing framework's capabilities. This is for applications that don't have HTTP easily available.

class `tapes.reporting.statsd.`**`StatsdReporter`**(*interval*, *host='localhost'*, *port=8125*, *prefix=None*, *registry=None*)
Reporter for StatsD.

class `tapes.reporting.stream.`**`ThreadedStreamReporter`**(*interval*, *stream=<open file '<stdout>', mode 'w'>*, *registry=None*)
Dumps JSON serialized metrics to a stream with an interval

class `tapes.reporting.tornado.`**`TornadoScheduledReporter`**(*interval*, *registry=None*, *io_loop=None*)
Scheduled reporter that uses a tornado IOLoop for scheduling

class `tapes.reporting.tornado.statsd.`**`TornadoStatsdReporter`**(*interval*, *host='localhost'*, *port=8125*, *prefix=None*, *registry=None*)
Reports to StatsD using an IOLoop for scheduling

class `tapes.reporting.tornado.stream.`**`TornadoStreamReporter`**(*interval*, *stream=<open file '<stdout>', mode 'w'>*, *registry=None*, *io_loop=None*)
Writes JSON serialized metrics to a stream using an `IOLoop` for scheduling

class `tapes.distributed.registry.`**`DistributedRegistry`**(*socket_addr='ipc://tapes_metrics.ipc'*)
A registry proxy that pushes metrics data to a `RegistryAggregator`.

`connect`()
Connects to the 0MQ socket and starts publishing.

class `tapes.distributed.registry.`**`RegistryAggregator`**(*reporter*, *socket_addr='ipc://tapes_metrics.ipc'*)
Aggregates multiple registry proxies and reports on the unified metrics.

`start`(*fork=True*)
Starts the registry aggregator.

Parameters **`fork`** – whether to fork a process; if `False`, blocks and stays in the existing process

`stop`()
Terminates the forked process.

Only valid if started as a fork, because... well you wouldn't get here otherwise. :return:

This is a native Python metrics library implementation. It currently supports Python 2.7, 3.4 and PyPy, but most other versions should probably work.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search