Statsd Metrics Documentation

Release 1.0.0

Farzad Ghanei

Aug 05, 2018

Contents

Contents:

CHAPTER 1

Metrics

Define the data types used in Statsd. Each data type is defined as a class, supported data types are:

- *Counter*
- *Timer*
- *Gauge*
- *Set*
- *GaugeDelta*

Note: The metric classes and helper functions are available from the package directly, but internally they are defined in *metrics* module. So there is no need to import the *metrics* module direcly, unless you're trying to access those objects that are not used reguraly and hence are not exported, like the *AbstractMetric* class.

Each metric requires a name and a value.

```
from statsdmetrics import Counter, Timer
counter = Counter('event.login', 1)
timer = Timer('db.query.user', 10)
```

An optional sample rate can be specified for the metrics. Sample rate is used by the client and the server to help to reduce network traffic, or reduce the load on the server.

```
>>> from statsdmetrics import Counter
>>> counter = Counter('event.login', 1, 0.2)
>>> counter.name
'event.login'
>>> counter.count
1
>>> counter.sample_rate
0.2
```

All metrics have *name* and *sample_rate* properties, but they store their value in different properties.

Metrics provide *to_request()* method to create the proper value used to send the metric to the server.

```
>>> from statsdmetrics import Counter, Timer, Gauge, Set, GaugeDelta
>>> counter = Counter('event.login', 1, 0.2)
>>> counter.to_request()
'event.login:1|c|@0.2'
>>> timer = Timer('db.query.user', 10, 0.5)
>>> timer.to_request()
'db.query.user:10|ms|@0.5'
>>> gauge = Gauge('memory', 20480)
>>> gauge.to_request()
'memory:20480|g'
>>> set_ = Set('unique.users', 'first')
>>> set_.to_request()
'unique.users:first|s'
>>> delta = GaugeDelta('memory', 128)
>>> delta.to_request()
'memory:+128|g'
>>> delta.delta = -256
>>> delta.to_request()
'memory:-256|g'
```

1.1 `metrics` – Metric classes and helper functions

1.1.1 Metric Classes

class metrics.**AbstractMetric**
Abstract class that all metric classes would extend from

name
the name of the metric

sample_rate
the rate of sampling that the client considers when sending metrics

to_request() → str
return the string that is used in the Statsd request to send the metric

class metrics.**Counter**(*name*, *count*[, *sample_rate*])
A metric to count events

count
current count of events being reporeted via the metric

class metrics.**Timer**(*name*, *milliseconds*[, *sample_rate*])
A metric for timing durations, in milliseconds.

milliseconds
number of milliseconds for the duration

class metrics.**Gauge**(*name*, *value*[, *sample_rate*])
Any arbitrary value, like the memory usage in bytes.

value
the value of the metric

class metrics.**Set**(*name*, *value*[, *sample_rate*])
A set of unique values counted on the server side for each sampling period. Techincally the value could be anything that can be serialized to a string (to be sent on the request).

value
the value of the metric

class metrics.**GaugeDelta**(*name*, *delta*[, *sample_rate*])
A value change in a gauge, could be a positive or negative numeric value.

delta
the difference in the value of the gauge

1.1.2 Module functions

metrics.**normalize_metric_name**(*name*) → str
normalize a metric name, removing characters that might not be welcome by common backends.

```
>>> from statsdmetrics import normalize_metric_name
>>> normalize_metric_name("will replace some, and $remove! others*")
'will_replace_some_and_remove_others'
```

If passed argument is not a string, an TypeError is raised.

metrics.**parse_metric_from_request**(*request*) → str
parse a metric object from a request string.

```
>>> from statsdmetrics import parse_metric_from_request
>>> metric = parse_metric_from_request("memory:2048|g")
>>> type(metric)
<class 'statsdmetrics.metrics.Gauge'>
>>> metric.name, metric.value, metric.sample_rate
('memory', 2048.0, 1)
>>> metric = parse_metric_from_request('event.connections:-2|c|@0.6')
>>> type(metric)
<class 'statsdmetrics.metrics.Counter'>
>>> metric.name, metric.count, metric.sample_rate
('event.connections', -2, 0.6)
```

If the request is invalid, a ValueError is raised.

CHAPTER 2

Client

To send the metrics to Statsd server, client classes are available in the *client* package and *client.tcp* module.

2.1 client – Statsd client

class client.**Client**(*host*, *port=8125*, *prefix=''*)

Default Statsd client that sends each metric in a separate UDP request

host

the host name (or IP address) of Statsd server. This property is **readonly**.

port

the port number of Statsd server. This property is **readonly**.

prefix

the default prefix for all metric names sent from the client

remote_address

tuple of resolved server address (host, port). This property is **readonly**.

increment(*name*, *count=1*, *rate=1*)

Increase a *Counter* metric by `count` with an integer value. An optional sample rate can be specified.

decrement(*name*, *count=1*, *rate=1*)

Decrease a *Counter* metric by `count` with an integer value. An optional sample rate can be specified.

timing(*name*, *milliseconds*, *rate=1*)

Send a *Timer* metric for the duration of a task in milliseconds. The `milliseconds` should be a none-negative numeric value. An optional sample rate can be specified.

gauge(*name*, *value*, *rate=1*)

Send a *Gauge* metric with the specified value. The `value` should be a none-negative numeric value. An optional sample rate can be specified.

set(*name*, *value*, *rate=1*)

Send a *Set* metric with the specified value. The server will count the number of unique values during each

sampling period. The `value` could be any value that can be converted to a string. An optional sample rate can be specified.

gauge_delta(*name*, *delta*, *rate=1*)
Send a *GaugeDelta* metric with the specified delta. The `delta` should be a numeric value. An optional sample rate can be specified.

batch_client(*size=512*)
Create a *BatchClient* object, using the same configurations of current client. This batch client could be used as a context manager in a `with` statement. After the `with` block when the context manager exits, all the metrics are flushed to the server in batch requests.

chronometer()
Create a *client.timing.Chronometer* that uses current client to send timing metrics.

stopwatch(*name*, *rate=1*, *reference=None*)
Create a *client.timing.Stopwatch* that uses current client to send timing metrics.

Note: Most Statsd servers do not apply the sample rate on timing metrics calculated results (mean, percentile, max, min), gauge or set metrics, but they take the rate into account for the number of received samples. Some statsd servers totally ignore the sample rate for metrics other than counters.

2.1.1 Examples

```
from statsdmetrics.client import Client
client = Client("stats.example.org")
client.increment("login")
client.timing("db.search.username", 3500)
client.prefix = "other"
client.gauge_delta("memory", -256)
client.decrement(name="connections", count=2)
```

```
from statsdmetrics.client import Client

client = Client("stats.example.org")
with client.batch_client() as batch_client:
    batch_client.increment("login")
    batch_client.decrement(name="connections", count=2)
    batch_client.timing("db.search.username", 3500)
# now all metrics are flushed automatically in batch requests
```

class client.**BatchClient**(*host*, *port=8125*, *prefix=''*, *batch_size=512*)
Statsd client that buffers all metrics and sends them in batch requests over UDP when instructed to flush the metrics explicitly.

Each UDP request might contain multiple metrics, but limited to a certain batch size to avoid UDP fragmentation.

The size of batch requests is not the fixed size of the requests, since metrics can not be broken into multiple requests. So if adding a new metric overflows this size, then that metric will be sent in a new batch request.

batch_size
Size of each batch request. This property is **readonly**.

clear()
Clear buffered metrics

flush()
Send the buffered metrics in batch requests.

unit_client()
Create a *Client* object, using the same configurations of current batch client to send the metrics on each request. The client uses the same resources as the batch client.

```
from statsdmetrics.client import BatchClient

client = BatchClient("stats.example.org")
client.set("unique.ip_address", "10.10.10.1")
client.gauge("memory", 20480)
client.flush() # sends one UDP packet to remote server, carrying both metrics
```

2.2 client.tcp – Statsd client sending metrics over TCP

class client.tcp.**TCPClient**(*host*, *port=8125*, *prefix=''*)
Statsd client that sends each metric in separate requests over TCP.

Provides the same interface as *Client*.

2.2.1 Examples

```
from statsdmetrics.client.tcp import TCPClient
client = TCPClient("stats.example.org")
client.increment("login")
client.timing("db.search.username", 3500)
client.prefix = "other"
client.gauge_delta("memory", -256)
client.decrement(name="connections", count=2)
```

```
from statsdmetrics.client.tcp import TCPClient

client = TCPClient("stats.example.org")
with client.batch_client() as batch_client:
    batch_client.increment("login")
    batch_client.decrement(name="connections", count=2)
    batch_client.timing("db.search.username", 3500)
# now all metrics are flushed automatically in batch requests
```

class client.tcp.**TCPBatchClient**(*host*, *port=8125*, *prefix=''*, *batch_size=512*)
Statsd client that buffers all metrics and sends them in batch requests over TCP when instructed to flush the metrics explicitly.

Provides the same interface as *BatchClient*.

```
from statsdmetrics.client.tcp import TCPBatchClient

client = TCPBatchClient("stats.example.org")
client.set("unique.ip_address", "10.10.10.1")
client.gauge("memory", 20480)
client.flush() # sends one TCP packet to remote server, carrying both metrics
```

CHAPTER 3

Timing Helpers

Classes to help measure time and send *metrics.Timer* metrics using any *client*.

3.1 client.timing – Timing helpers

class client.timing.**Chronometer**(*client*, *rate=1*)

Chronometer calculates duration (of function calls, etc.) and sends them with provided metric names. Normally these is no need to instantiate this class directly, but you can call *client.Client.chronometer()* on any client, to get a configured Chronometer.

client

The client used to send the timing metrics. This can be any client from *client* package.

rate

the default sample rate for metrics to send. Should be a float between 0 and 1. This is the same as used in all clients.

since(*name*, *timestamp*, *rate=None*)

Calculate the time passed since the given timestamp, and send a *Timer* metric with the provided name. The timestamp can be a float (seconds passed from epoch, as returned by `time.time()`, or a `datetime.datetime` instance. Rate is the sample rate to use, or None to use the default sample rate of the Chronometer.

time_callable(*name*, *target*, *rate=None*, *args=()*, *kwargs={}*)

Calculate the time it takes to run the callable *target* (with provided args and kwargs) and send the a *Timer* metric with the specific name. Rate is the sample rate to use, or None to use the default sample rate of the Chronometer.

wrap(*name*, *rate=None*)

Used as a function decorator, to calculate the time it takes to run the decorated function, and send a *Timer* metric with the specified name. Rate is the sample rate to use, or None to use the default sample rate of the Chronometer.

3.1.1 Examples

```
from time import time, sleep
from statsdmetrics.client import Client
from statsdmetrics.client.timing import Chronometer

start_time = time()
client = Client("stats.example.org")
chronometer = Chronometer(client)
chronometer.since("instantiate", start_time)

def wait(secs):
    sleep(secs) # or any timed operation

chronometer.time_callable("waited", wait, args=(0.56,))

@chronometer.wrap("wait_decorated")
def another_wait(secs):
    sleep(secs) # or any timed operation

another_wait(0.23) # sends the "wait_decorated" Timer metric
chronometer.since("overall", start_time)
```

If a batch client (like *client.BatchClient* or *client.tcp.TCPBatchClient*) is used, then the behavior of the client requires an explicit *flush()* call.

```
from datetime import datetime
from statsdmetrics.client.tcp import TCPBatchCPClient
from statsdmetrics.client.timing import Chronometer

start_time = datetime.now()
client = TCPBatchClient("stats.example.org")
chronometer = Chronometer(client)
chronometer.since("instantiate", start_time)

def wait_with_kwargs(name, key=val):
    sleep(1) # or any timed operation

chronometer.time_callable("waited", wait_with_kwargs, kwargs=dict(name="foo", key="bar
↪"))
client.flush()
```

class client.timing.**Stopwatch**(*client*, *name*, *rate=1*, *reference=None*)

Stopwatch calculates duration passed from a given reference time (by default uses the instantiation time) for a specific metric name. So time passed since the reference time can be sent multiple times. Normally these is no need to instantiate this class directly, but you can call *client.Client.stopwatch()* on any client, to get a configured Chronometer.

client

The client used to send the timing metrics. This can be any client from *client* package.

name

The name for the metric sent by the stopwatch.

rate

The default sample rate for metrics to send. Should be a float between 0 and 1. This is the same as used in all clients.

reference

The time reference that duration is calculated from. It's a float value of seconds passed since epoch, same as *time.time()*.

reset()
Reset the stopwatch, updating the reference with current time. Returns a self reference for method chaining.

send(*rate=None*)
Calculate time passed since *reference* and send the metric. A sampling rate can be specified, or *None* (default) uses the default sampling rate of the stopwatch. Returns a self reference for method chaining.

3.1.2 Examples

```
from time import time, sleep
from statsdmetrics.client import Client
from statsdmetrics.client.timing import Stopwatch

start_time = time()
client = Client("stats.example.org")
stopwatch = Stopwatch(client, "process", start_time)

sleep(2) # do stuff
stopwatch.send()
sleep(1) # do other stuff
stopwatch.send()
```

If a batch client (like *client.BatchClient* or *client.tcp.TCPBatchClient*) is used, then the behavior of the client requires an explicit *flush()* call.

```
from datetime import datetime
from statsdmetrics.client.tcp import TCPBatchCPClient
from statsdmetrics.client.timing import Stopwatch

start_time = time()
client = TCPBatchClient("stats.example.org")
stopwatch = Stopwatch(client, "process", start_time)

sleep(3) # do stuff
stopwatch.send()
sleep(1) # do other stuff
stopwatch.send()

client.flush()
```

Stopwatch is a context manager, so can be used to measure duration of a *with* block

```
from time import time, sleep
from statsdmetrics.client import Client
from statsdmetrics.client.timing import Stopwatch

client = Client("stats.example.org")
with client.stopwatch("some_block"):
    sleep(3) # do stuff in the context

# now a Timer metric named "some_block" is sent, whose value is the duration of the␣
↪block
```

CHAPTER 4

Introduction

Metric classes for Statsd, and Statsd clients (each metric in a single request, or send batch requests). Provides APIs to create, parse or send metrics to a Statsd server.

The library also comes with a rich set of Statsd clients using the same metric classes, and Python standard library socket module.

4.1 Metric Classes

Metric classes represent the data used in Statsd protocol excluding the IO, to create, represent and parse Statsd requests. So any Statsd server and client regardless of the IO implementation can use them to send/receive Statsd requests.

Available metrics:

- *Counter*
- *Timer*
- *Gauge*
- *Set*
- *GaugeDelta*

The *metrics* module also provides helper functions to normalize metric names, and a parse a Statsd request and return the corresponding metric object. This could be used on the server side to parse the received requests.

4.2 Clients

A rich set of Statsd clients using the same metric classes, and Python standard library socket module.

- *Client*: Default client, sends request on each call using UDP
- *BatchClient*: Buffers metrics and flushes them in batch requests using UDP

- *TCPClient*: Sends request on each call using TCP
- *TCPBatchClient*: Buffers metrics and flushes them in batch requests using TCP

4.3 Timing Helpers

- *Chronometer*: Measure duration and send multiple *Timer* metrics
- *Stopwatch*: Measure time passed from a given reference and send *Timer* metrics with a specific name

CHAPTER 5

Installation

```
$ pip install statsdmetrics
```

5.1 Dependencies

The only dependencies are Python 2.7+ and setuptools. CPython 2.7, 3.4+, 3.7-dev, PyPy and Jython are tested)

However on development (and test) environment pytest, mock is required (for Python 2), typing is recommended.

```
# on dev/test env
$ pip install -r requirements-dev.txt
```

CHAPTER 6

License

Statsd metrics is released under the terms of the MIT license.

CHAPTER 7

Development

- Code is on GitHub
- Documentations are on Read The Docs

Python Module Index

c

m

Index

U

W