StatsD client for Ruby apps

This is a ruby client for statsd (https://github.com/statsd/statsd). It provides a lightweight way to track and measure metrics in your application.

We call out to statsd by sending data over a UDP socket. UDP sockets are fast, but unreliable, there is no guarantee that your data will ever arrive at its location. In other words, fire and forget. This is perfect for this use case because it means your code doesn't get bogged down trying to log statistics. We send data to statsd several times per request and haven't noticed a performance hit.

For more information about StatsD, see the README of the StatsD project.

Configuration

It's recommended to configure this library by setting environment variables. The following environment variables are supported:

- `STATSD_ADDR`: (default `localhost:8125`) The address to send the StatsD UDP datagrams to.
- `STATSD_IMPLEMENTATION`: (default: `datadog`). The StatsD implementation you are using. `statsd` and `datadog` are supported. Some features are only available on certain implementations,
- `STATSD_ENV`: The environment StatsD will run in. If this is not set explicitly, this will be determined based on other environment variables, like `RAILS_ENV` or `ENV`. The library will behave differently:
  - In the **production** and **staging** environment, the library will actually send UDP packets.
  - In the **test** environment, it will swallow all calls, but allows you to capture them for testing purposes. See below for notes on writing tests.
  - In **development** and all other environments, it will write all calls to the log (`StatsD.logger`, which by default writes to STDOUT).
- `STATSD_SAMPLE_RATE`: (default: `1.0`) The default sample rate to use for all metrics. This can be used to reduce the amount of network traffic and CPU overhead the usage of this library generates. This can be overridden in a metric method call.
- `STATSD_PREFIX`: The prefix to apply to all metric names. This can be overridden in a metric method call.
- `STATSD_DEFAULT_TAGS`: A comma-separated list of tags to apply to all metrics. (Note: tags are not supported by all implementations.)

- `STATSD_BUFFER_CAPACITY`: (default: `5000`) The maximum amount of events that may be buffered before emitting threads will start to block. Increasing this value may help for application generating spikes of events. However if the application emit events faster than they can be sent, increasing it won't help. If set to `0`, batching will be disabled, and events will be sent in individual UDP packets, which is much slower.
- `STATSD_FLUSH_INTERVAL`: (default: `1`) Deprecated. Setting this to `0` is equivalent to setting `STATSD_BUFFER_CAPACITY` to `0`.
- `STATSD_MAX_PACKET_SIZE`: (default: `1472`) The maximum size of UDP packets. If your network is properly configured to handle larger packets you may try to increase this value for better performance, but most network can't handle larger packets.

StatsD keys

StatsD keys look like 'admin.logins.api.success'. Dots are used as namespace separators.

Usage

You can either use the basic methods to submit stats over StatsD, or you can use the metaprogramming methods to instrument your methods with some basic stats (call counts, successes & failures, and timings).

StatsD.measure Lets you benchmark how long the execution of a specific method takes.

```ruby
# You can pass a key and a ms value
StatsD.measure('GoogleBase.insert', 2.55)

# or more commonly pass a block that calls your code
StatsD.measure('GoogleBase.insert') do
  GoogleBase.insert(product)
end
```

StatsD.increment Lets you increment a key in statsd to keep a count of something. If the specified key doesn't exist it will create it for you.

```ruby
# increments default to +1
StatsD.increment('GoogleBase.insert')
# you can also specify how much to increment the key by
StatsD.increment('GoogleBase.insert', 10)
# you can also specify a sample rate, so only 1/10 of events
# actually get to statsd. Useful for very high volume data
```

```
StatsD.increment('GoogleBase.insert', sample_rate: 0.1)
```

StatsD.gauge A gauge is a single numerical value that tells you the state of the system at a point in time. A good example would be the number of messages in a queue.

```
StatsD.gauge('GoogleBase.queued', 12, sample_rate: 1.0)
```

Normally, you shouldn't update this value too often, and therefore there is no need to sample this kind metric.

StatsD.set A set keeps track of the number of unique values that have been seen. This is a good fit for keeping track of the number of unique visitors. The value can be a string.

```
# Submit the customer ID to the set. It will only be counted if it hasn't been seen before.
StatsD.set('GoogleBase.customers', "12345", sample_rate: 1.0)
```

Because you are counting unique values, the results of using a sampling value less than 1.0 can lead to unexpected, hard to interpret results.

StatsD.histogram Builds a histogram of numeric values.

```

StatsD.histogram('Order.value', order.value_in_usd.to_f, tags: { source: 'POS' })
```

Because you are counting unique values, the results of using a sampling value less than 1.0 can lead to unexpected, hard to interpret results.

Note: This is only supported by the beta datadog implementation.

StatsD.distribution A modified gauge that submits a distribution of values over a sample period. Arithmetic and statistical calculations (percentiles, average, etc.) on the data set are performed server side rather than client side like a histogram.

```
StatsD.distribution('shipit.redis_connection', 3)
```

Note: This is only supported by the beta datadog implementation.

StatsD.event An event is a (title, text) tuple that can be used to correlate metrics with something that occurred within the system. This is a good fit for instance to correlate response time variation with a deploy of the new code.

```
StatsD.event('shipit.deploy', 'started')
```

Note: This is only supported by the datadog implementation.

Events support additional metadata such as `date_happened`, `hostname`, `aggregation_key`, `priority`, `source_type_name`, `alert_type`.

StatsD.service_check An event is a (check_name, status) tuple that can be used to monitor the status of services your application depends on.

```
StatsD.service_check('shipit.redis_connection', 'ok')
```

Note: This is only supported by the datadog implementation.

Service checks support additional metadata such as `timestamp`, `hostname`, `message`.

Metaprogramming Methods

As mentioned, it's most common to use the provided metaprogramming methods. This lets you define all of your instrumentation in one file and not litter your code with instrumentation details. You should enable a class for instrumentation by extending it with the `StatsD::Instrument` class.

```
GoogleBase.extend StatsD::Instrument
```

Then use the methods provided below to instrument methods in your class.

statsd_measure This will measure how long a method takes to run, and submits the result to the given key.

```
GoogleBase.statsd_measure :insert, 'GoogleBase.insert'
```

statsd_count This will increment the given key even if the method doesn't finish (ie. raises).

```
GoogleBase.statsd_count :insert, 'GoogleBase.insert'
```

Note how I used the 'GoogleBase.insert' key above when measuring this method, and I reused here when counting the method calls. StatsD automatically separates these two kinds of stats into namespaces so there won't be a key collision here.

statsd_count_if This will only increment the given key if the method executes successfully.

```ruby
GoogleBase.statsd_count_if :insert, 'GoogleBase.insert'
```

So now, if GoogleBase#insert raises an exception or returns false (ie. result == false), we won't increment the key. If you want to define what success means for a given method you can pass a block that takes the result of the method.

```ruby
GoogleBase.statsd_count_if :insert, 'GoogleBase.insert' do |response|
  response.code == 200
end
```

In the above example we will only increment the key in statsd if the result of the block returns true. So the method is returning a Net::HTTP response and we're checking the status code.

statsd_count_success Similar to statsd_count_if, except this will increment one key in the case of success and another key in the case of failure.

```ruby
GoogleBase.statsd_count_success :insert, 'GoogleBase.insert'
```

So if this method fails execution (raises or returns false) we'll increment the failure key ('GoogleBase.insert.failure'), otherwise we'll increment the success key ('GoogleBase.insert.success'). Notice that we're modifying the given key before sending it to statsd.

Again you can pass a block to define what success means.

```ruby
GoogleBase.statsd_count_success :insert, 'GoogleBase.insert' do |
    response|
  response.code == 200
end
```

Instrumenting Class Methods

You can instrument class methods, just like instance methods, using the metaprogramming methods. You simply have to configure the instrumentation on the singleton class of the Class you want to instrument.

```ruby
AWS::S3::Base.singleton_class.statsd_measure :request, 'S3.request'
```

Dynamic Metric Names

You can use a lambda function instead of a string dynamically set the name of the metric. The lambda function must accept two arguments: the object the function is being called on and the array of arguments passed.

```
GoogleBase.statsd_count :insert, lambda{|object, args| object.class.
    to_s.downcase + "." + args.first.to_s + ".insert" }
```

Tags

The Datadog implementation supports tags, which you can use to slice and dice metrics in their UI. You can specify a list of tags as an option, either standalone tag (e.g. `"mytag"`), or key value based, separated by a colon: `"env:production"`.

```
StatsD.increment('my.counter', tags: ['env:production', 'unicorn'])
GoogleBase.statsd_count :insert, 'GoogleBase.insert', tags: ['env:
    production']
```

If implementation is not set to `:datadog`, tags will not be included in the UDP packets, and a warning is logged to `StatsD.logger`.

You can use lambda function that instead of a list of tags to set the metric tags. Like the dynamic metric name, the lambda function must accept two arguments: the object the function is being called on and the array of arguments passed.

```
metric_tagger = lambda { |object, args| { "key": args.first } }
GoogleBase.statsd_count(:insert, 'GoogleBase.insert', tags:
    metric_tagger)
```

> You can only use the dynamic tag while using the instrumentation through metaprogramming methods

Testing

This library comes with a module called `StatsD::Instrument::Assertions` and `StatsD::Instrument::Matchers` to help you write tests to verify StatsD is called properly.

minitest

```
class MyTestcase < Minitest::Test
  include StatsD::Instrument::Assertions

  def test_some_metrics
    # This will pass if there is exactly one matching StatsD call
    # it will ignore any other, non matching calls.
    assert_statsd_increment('counter.name', sample_rate: 1.0) do
```

```
      StatsD.increment('unrelated') # doesn't match
      StatsD.increment('counter.name', sample_rate: 1.0) # matches
      StatsD.increment('counter.name', sample_rate: 0.1) # doesn't match
    end

    # Set `times` if there will be multiple matches:
    assert_statsd_increment('counter.name', times: 2) do
      StatsD.increment('unrelated') # doesn't match
      StatsD.increment('counter.name', sample_rate: 1.0) # matches
      StatsD.increment('counter.name', sample_rate: 0.1) # matches too
    end
  end

  def test_no_udp_traffic
    # Verifies no StatsD calls occurred at all.
    assert_no_statsd_calls do
      do_some_work
    end

    # Verifies no StatsD calls occurred for the given metric.
    assert_no_statsd_calls('metric_name') do
      do_some_work
    end
  end

  def test_more_complicated_stuff
    # capture_statsd_calls will capture all the StatsD calls in the
    # given block, and returns them as an array. You can then run your
    # own assertions on it.
    metrics = capture_statsd_calls do
      StatsD.increment('mycounter', sample_rate: 0.01)
    end

    assert_equal 1, metrics.length
    assert_equal 'mycounter', metrics[0].name
    assert_equal :c, metrics[0].type
    assert_equal 1, metrics[0].value
    assert_equal 0.01, metrics[0].sample_rate
  end
end
```

RSpec

```
RSpec.configure do |config|
  config.include StatsD::Instrument::Matchers
end

RSpec.describe 'Matchers' do
```

```ruby
  context 'trigger_statsd_increment' do
    it 'will pass if there is exactly one matching StatsD call' do
      expect { StatsD.increment('counter') }.to trigger_statsd_increment('counter')
    end

    it 'will pass if it matches the correct number of times' do
      expect {
        2.times do
          StatsD.increment('counter')
        end
      }.to trigger_statsd_increment('counter', times: 2)
    end

    it 'will pass if it matches argument' do
      expect {
        StatsD.measure('counter', 0.3001)
      }.to trigger_statsd_measure('counter', value: be_between(0.29, 0.31))
    end

    it 'will pass if there is no matching StatsD call on negative expectation' do
      expect { StatsD.increment('other_counter') }.not_to trigger_statsd_increment('counter')
    end

    it 'will pass if every statsD call matches its call tag variations' do
      expect do
        StatsD.increment('counter', tags: ['variation:a'])
        StatsD.increment('counter', tags: ['variation:b'])
      end.to trigger_statsd_increment('counter', times: 1, tags: ["variation:a"]).and trigger_statsd_increment('counter', times: 1, tags: ["variation:b"])
    end
  end
end
```

Notes

Compatibility

The library is tested against Ruby 2.3 and higher. We are not testing on different Ruby implementations besides MRI, but we expect it to work on other implementations as well.

Reliance on DNS

Out of the box StatsD is set up to be unidirectional fire-and-forget over UDP. Configuring the StatsD host to be a non-ip will trigger a DNS lookup (i.e. a synchronous TCP round trip). This can be particularly problematic in clouds that have a shared DNS infrastructure such as AWS.

1. Using a hardcoded IP avoids the DNS lookup but generally requires an application deploy to change.
2. Hardcoding the DNS/IP pair in /etc/hosts allows the IP to change without redeploying your application but fails to scale as the number of servers increases.
3. Installing caching software such as nscd that uses the DNS TTL avoids most DNS lookups but makes the exact moment of change indeterminate.

Links

This library was developed for shopify.com and is MIT licensed.

- API documentation
- The changelog covers the changes between releases.
- Contributing notes if you are interested in contributing to this library.