OSprofiler Documentation

Release

OpenStack Foundation

March 24, 2016

Contents

OSProfiler is an OpenStack cross-project profiling library.

CHAPTER 1

Background

OpenStack consists of multiple projects. Each project, in turn, is composed of multiple services. To process some request, e.g. to boot a virtual machine, OpenStack uses multiple services from different projects. In the case something works too slowly, it's extremely complicated to understand what exactly goes wrong and to locate the bottleneck.

To resolve this issue, we introduce a tiny but powerful library, **osprofiler**, that is going to be used by all OpenStack projects and their python clients. To be able to generate 1 trace per request, that goes through all involved services, and builds a tree of calls (see an example).

CHAPTER 2

Why not cProfile and etc?

The scope of this library is quite different:

- We are interested in getting one trace of points from different service, not tracing all python calls inside one process.
- This library should be easy integratable in OpenStack. This means that:
  - It shouldn't require too many changes in code bases of integrating projects.
  - We should be able to turn it off fully.
  - We should be able to keep it turned on in lazy mode in production (e.g. admin should be able to "trace" on request).

CHAPTER 3

OSprofiler API

There are a couple of things that you should know about API before using it.

- **4 ways to add a new trace point**

```
from osprofiler import profiler

def some_func():
    profiler.start("point_name", {"any_key": "with_any_value"})
    # your code
    profiler.stop({"any_info_about_point": "in_this_dict"})


@profiler.trace("point_name",
                info={"any_info_about_point": "in_this_dict"},
                hide_args=False)
def some_func2(*args, **kwargs):
    # If you need to hide args in profile info, put hide_args=True
    pass

def some_func3():
    with profiler.Trace("point_name",
                        info={"any_key": "with_any_value"}):
        # some code here

@profiler.trace_cls("point_name", info={}, hide_args=False,
                    trace_private=False)
class TracedClass(object):

    def traced_method(self):
        pass

    def _traced_only_if_trace_private_true(self):
         pass
```

- **How profiler works?**
  - **@profiler.Trace()** and **profiler.trace()** are just syntax sugar, that just calls **profiler.start()** & **profiler.stop()** methods.
  - Every call of **profiler.start()** & **profiler.stop()** sends to **collector** 1 message. It means that every trace point creates 2 records in the collector. *(more about collector & records later)*
  - Nested trace points are supported. The sample below produces 2 trace points:

```
profiler.start("parent_point")
profiler.start("child_point")
profiler.stop()
profiler.stop()
```

The implementation is quite simple. Profiler has one stack that contains ids of all trace points. E.g.:

```
profiler.start("parent_point") # trace_stack.push(<new_uuid>)
                               # send to collector -> trace_stack[-2:]

profiler.start("parent_point") # trace_stack.push(<new_uuid>)
                               # send to collector -> trace_stack[-2:]
profiler.stop()                # send to collector -> trace_stack[-2:]
                               # trace_stack.pop()

profiler.stop()                # send to collector -> trace_stack[-2:]
                               # trace_stack.pop()
```

It's simple to build a tree of nested trace points, having **(parent_id, point_id)** of all trace points.

- **Process of sending to collector**

  Trace points contain 2 messages (start and stop). Messages like below are sent to a collector:

```
{
    ``name'': <point_name>-(start|stop)
    ``base_id'': <uuid>,
    ``parent_id'': <uuid>,
    ``trace_id'': <uuid>,
    ``info'': <dict>
}

* base_id - <uuid> that is equal for all trace points that belong
            to one trace, this is done to simplify the process of retrieving
            all trace points related to one trace from collector
* parent_id - <uuid> of parent trace point
* trace_id - <uuid> of current trace point
* info - the dictionary that contains user information passed when calling
         profiler start() & stop() methods.
```

- **Setting up the collector.**

  The profiler doesn't include a trace point collector. The user/developer should instead provide a method that sends messages to a collector. Let's take a look at a trivial sample, where the collector is just a file:

```
import json

from osprofiler import notifier

def send_info_to_file_collector(info, context=None):
    with open("traces", "a") as f:
        f.write(json.dumps(info))

notifier.set(send_info_to_file_collector)
```

So now on every **profiler.start()** and **profiler.stop()** call we will write info about the trace point to the end of the **traces** file.

- **Initialization of profiler.**

  If profiler is not initialized, all calls to **profiler.start()** and **profiler.stop()** will be ignored.

  Initialization is a quite simple procedure.

```
from osprofiler import profiler

profiler.init("SECRET_HMAC_KEY", base_id=<uuid>, parent_id=<uuid>)
```

  `SECRET_HMAC_KEY` - will be discussed later, because it's related to the integration of OSprofiler & OpenStack.

  base_id and **trace_id** will be used to initialize stack_trace in profiler, e.g. stack_trace = [base_id, trace_id].

- **OSProfiler CLI.**

  To make it easier for end users to work with profiler from CLI, osprofiler has entry point that allows them to retrieve information about traces and present it in human readable from.

  Available commands:

  - Help message with all available commands and their arguments:

```
$ osprofiler -h/--help
```

  - OSProfiler version:

```
$ osprofiler -v/--version
```

  - Results of profiling can be obtained in JSON (option: `--json`) and HTML (option: `--html`) formats:

```
$ osprofiler trace show <trace_id> --json/--html
```

  hint: option `--out` will redirect result of `osprofiler trace show` in specified file:

```
$ osprofiler trace show <trace_id> --json/--html --out /path/to/file
```

Integration with OpenStack

There are 4 topics related to integration OSprofiler & OpenStack:

- **What we should use as a centralized collector?**

  We decided to use Ceilometer, because:

  - It's already integrated in OpenStack, so it's quite simple to send notifications to it from all projects.
  - There is an OpenStack API in Ceilometer that allows us to retrieve all messages related to one trace. Take a look at *osprofiler.parsers.ceilometer:get_notifications*

- **How to setup profiler notifier?**

  We decided to use olso.messaging Notifier API, because:

  - oslo.messaging is integrated in all projects
  - It's the simplest way to send notification to Ceilometer, take a look at: *osprofiler.notifiers.messaging.Messaging:notify* method
  - We don't need to add any new CONF options in projects

- **How to initialize profiler, to get one trace across all services?**

  To enable cross service profiling we actually need to do send from caller to callee (base_id & trace_id). So callee will be able to init its profiler with these values.

  In case of OpenStack there are 2 kinds of interaction between 2 services:

  - REST API

    It's well known that there are python clients for every project, that generate proper HTTP requests, and parse responses to objects.

    These python clients are used in 2 cases:

    - User access -> OpenStack
    - Service from Project 1 would like to access Service from Project 2

    So what we need is to:

    - Put in python clients headers with trace info (if profiler is inited)
    - Add OSprofiler WSGI middleware to your service, this initializes the profiler, if and only if there are special trace headers, that are signed by one of the HMAC keys from api-paste.ini (if multiple keys exist the signing process will continue to use the key that was accepted during validation).

· The common items that are used to configure the middleware are the following (these can be provided when initializing the middleware object or when setting up the api-paste.ini file):

```
hmac_keys = KEY1, KEY2 (can be a single key as well)
```

Actually the algorithm is a bit more complex. The Python client will also sign the trace info with a HMAC key (lets call that key `A`) passed to profiler.init, and on reception the WSGI middleware will check that it's signed with *one of* the HMAC keys (the wsgi server should have key `A` as well, but may also have keys `B` and `C`) that are specified in api-paste.ini. This ensures that only the user that knows the HMAC key `A` in api-paste.ini can init a profiler properly and send trace info that will be actually processed. This ensures that trace info that is sent in that does **not** pass the HMAC validation will be discarded. **NOTE:** The application of many possible *validation* keys makes it possible to roll out a key upgrade in a non-impactful manner (by adding a key into the list and rolling out that change and then removing the older key at some time in the future).

- RPC API

  RPC calls are used for interaction between services of one project. It's well known that projects are using oslo.messaging to deal with RPC. It's very good, because projects deal with RPC in similar way.

  So there are 2 required changes:

  * On callee side put in request context trace info (if profiler was initialized)
  * On caller side initialize profiler, if there is trace info in request context.
  * Trace all methods of callee API (can be done via profiler.trace_cls).

- **What points should be tracked by default?**

  I think that for all projects we should include by default 5 kinds of points:

  - All HTTP calls - helps to get information about: what HTTP requests were done, duration of calls (latency of service), information about projects involved in request.
  - All RPC calls - helps to understand duration of parts of request related to different services in one project. This information is essential to understand which service produce the bottleneck.
  - All DB API calls - in some cases slow DB query can produce bottleneck. So it's quite useful to track how much time request spend in DB layer.
  - All driver calls - in case of nova, cinder and others we have vendor drivers. Duration
  - ALL SQL requests (turned off by default, because it produce a lot of traffic)