CONTENTS

1 What is the purpose of the project and vision for it? 3

2 Table of contents 5
   2.1 Initial setup ................................................................. 5
   2.2 Setting up the project ....................................................... 5
   2.3 Agent .............................................................................. 6
   2.4 Plugins ............................................................................. 6
   2.5 Tests ............................................................................... 7

3 Indices and tables 9
The *ceilometer* project aims to deliver a unique point of contact for billing systems to acquire all of the measurements they need to establish customer billing, across all current OpenStack core components with work underway to support future OpenStack components.
WHAT IS THE PURPOSE OF THE PROJECT AND VISION FOR IT?

- Provide efficient collection of metering data, in terms of CPU and network costs.
- Allow deployers to integrate with the metering system directly or by replacing components.
- Data may be collected by monitoring notifications sent from existing services or by polling the infrastructure.
- Allow deployers to configure the type of data collected to meet their operating requirements.
- The data collected by the metering system is made visible to some users through a REST API.
- The metering messages are signed and non-repudiable.

This documentation offers information on ceilometer works and how to contribute to the project.
2.1 System Architecture

2.1.1 High Level Description

There are 4 basic components to the system:

1. An agent runs on each compute node and polls for resource utilization statistics. There may be other types of agents in the future, but for now we will focus on creating the compute agent.

2. The collector runs on one or more central management servers to monitor the message queues (for notifications and for metering data coming from the agent). Notification messages are processed and turned into metering messages and sent back out onto the message bus using the appropriate topic. Metering messages are written to the data store without modification.

3. The data store is a database capable of handling concurrent writes (from one or more collector instances) and reads (from the API server).

4. The API server runs on one or more central management servers to provide access to the data from the data store. See EfficientMetering#API for details.

These services communicate using the standard OpenStack messaging bus. Only the collector and API server have access to the data store.

2.1.2 Detailed Description

Warning: These details cover only the compute agent and collector, as well as their communication via the messaging bus. More work is needed before the data store and API server designs can be documented.

Plugins

Although we have described a list of the metrics ceilometer should collect, we cannot predict all of the ways deployers will want to measure the resources their customers use. This means that ceilometer needs to be easy to extend and configure so it can be tuned for each installation. A plugin system based on setuptools entry points makes it easy to add new monitors in the collector or subagents for polling.

Each daemon provides basic essential services in a framework to be shared by the plugins, and the plugins do the specialized work. As a general rule, the plugins are asked to do as little work as possible. This makes them more efficient as greenlets, maximizes code reuse, and makes them simpler to implement.
Installing a plugin automatically activates it the next time the ceilometer daemon starts. A global configuration option can be used to disable installed plugins (for example, one or more of the “default” set of plugins provided as part of the ceilometer package).

Plugins may require configuration options, so when the plugin is loaded it is asked to add options to the global flags object, and the results are made available to the plugin before it is asked to do any work.

Rather than running and reporting errors or simply consuming cycles for no-ops, plugins may disable themselves at runtime based on configuration settings defined by other components (for example, the plugin for polling libvirt does not run if it sees that the system is configured using some other virtualization tool). The plugin is asked once at startup, after it has been loaded and given the configuration settings, if it should be enabled. Plugins should not define their own flags for enabling or disabling themselves.

**Warning:** Plugin self-deactivation is not implemented, yet.

Each plugin API is defined by the namespace and an abstract base class for the plugin instances. Plugins are not required to subclass from the API definition class, but it is encouraged as a way to discover API changes.

**Note:** There is ongoing work to add a generic plugin system to Nova. If that is implemented as part of the common library, ceilometer may use it (or adapt it as necessary for our use). If it remains part of Nova for Folsom we should probably not depend on it because loading plugins is trivial with setuptools.

### Polling

Metering data comes from two sources: through notifications built into the existing OpenStack components and by polling the infrastructure (such as via libvirt). Polling for compute resources is handled by an agent running on the compute node (where communication with the hypervisor is more efficient).

**Note:** We only poll compute resources for now, but when other types of polling are implemented the pollsters are likely to run somewhere other than the compute node.

The agent daemon is configured to run one or more *pollster* plugins using the `ceilometer.poll.compute` namespace. The agent periodically asks each pollster for instances of `Counter` objects. The agent framework converts the Counters to metering messages, which it then signs and transmits on the metering message bus.

The pollster plugins do not communicate with the message bus directly, unless it is necessary to do so in order to collect the information for which they are polling.

All polling happens with the same frequency, controlled by a global setting for the agent.

### Handling Notifications

The heart of the system is the collector, which monitors the message bus for data being provided by the pollsters via the agent as well as notification messages from other OpenStack components such as nova, glance, quantum, and swift.

The collector loads one or more *listener* plugins, using a namespace under `ceilometer.collector`. The namespace controls the exchange and topic where the listener is subscribed. For example, `ceilometer.collector.compute` listens on the `nova` exchange to the `notifications.info` topic while `ceilometer.collector.image` listens on the `glance` exchange for `notifications.info`.

The plugin provides a method to list the event types it wants and a callback for processing incoming messages. The registered name of the callback is used to enable or disable it using the global configuration option of the collector daemon. The incoming messages are filtered based on their event type value before being passed to the call-
Ceilometer Documentation, Release 0.0

back so the plugin only receives events it has expressed an interest in seeing. For example, a callback asking for 
compute.instance.create.end events under ceilometer.collector.compute would be invoked for those notification events on the nova exchange using the notifications.info topic.

The listener plugin returns an iterable with zero or more Counter instances based on the data in the incoming message. The collector framework code converts the Counter instances to metering messages and publishes them on the metering message bus. Although ceilometer includes a default storage solution to work with the API service, by republishing on the metering message bus we can support installations that want to handle their own data storage.

Handling Metering Messages

The listener for metering messages also runs in the collector daemon. It validates the incoming data and (if the signature is valid) then writes the messages to the data store.

Note: Because this listener uses openstack.common.rpc instead of notifications, it is implemented directly in the collector code instead of as a plugin.

Metering messages are signed using the hmac module in Python’s standard library. A shared secret value can be provided in the ceilometer configuration settings. The messages are signed by feeding the message key names and values into the signature generator in sorted order. Non-string values are converted to unicode and then encoded as UTF-8. The message signature is included in the message for verification by the collector, and stored in the database for future verification by consumers who access the data via the API.

RPC

Ceilometer uses openstack.common.rpc to cast messages from the agent to the collector.

See Also:

- http://wiki.openstack.org/EfficientMetering/ArchitectureProposalV1
- http://wiki.openstack.org/EfficientMetering#Architecture
- Bug 1010037 : allow different polling interval for each pollster

2.2 Measurements

A list of the current usage data that ceilometer will (eventually) listen for can be found at:


For a list of the planned meters, refer to:


Todo

Replicate the list of meters here.
2.3 Installing and Running the Development Version

Ceilometer has two daemons. The agent runs on the Nova compute node(s) and the collector runs on the cloud’s management node(s). In a development environment created by devstack, these two are typically the same server. They do not have to be, though, so some of the instructions below are duplicated. Skip the steps you have already done.

2.3.1 Installing the Collector

1. Install and configure nova.

   The collector daemon imports code from nova, so it needs to be run on a server where nova has already been installed.

   **Note:** Ceilometer makes extensive use of the messaging bus, but has not yet been tested with ZeroMQ. We recommend using Rabbit or qpid for now.

2. Install MongoDB.

   Follow the instructions to install the MongoDB package for your operating system, then start the service.

3. Clone the ceilometer git repository to the management server:

   ```
   $ cd /opt/stack
   $ git clone https://github.com/stackforge/ceilometer.git
   ```

4. As a user with root permissions or sudo privileges, run the ceilometer installer:

   ```
   $ cd ceilometer
   $ sudo python setup.py install
   ```

5. Configure ceilometer.

   Ceilometer needs to know about some of the nova configuration options, so the simplest way to start is copying /etc/nova/nova.conf to /etc/ceilometer-collector.conf. Some of the logging settings used in nova break ceilometer, so they need to be removed. For example, as a user with root permissions:

   ```
   $ grep -v format_string /etc/nova/nova.conf > /etc/ceilometer-collector.conf
   ```

   Refer to *Configuration Options* for details about any other options you might want to modify before starting the service.

6. Start the collector.

   ```
   $ ./bin/ceilometer-collector
   ```

   **Note:** The default development configuration of the collector logs to stderr, so you may want to run this step using a screen session or other tool for maintaining a long-running program in the background.

2.3.2 Installing the Compute Agent

**Note:** The compute agent must be installed on each nova compute node.
1. Install and configure nova.

The collector daemon imports code from nova, so it needs to be run on a server where nova has already been installed.

Note: Ceilometer makes extensive use of the messaging bus, but has not yet been tested with ZeroMQ. We recommend using Rabbit or qpid for now.

2. Clone the ceilometer git repository to the server:

$ cd /opt/stack
$ git clone https://github.com/stackforge/ceilometer.git

4. As a user with root permissions or sudo privileges, run the ceilometer installer:

$ cd ceilometer
$ sudo python setup.py install

5. Configure ceilometer.

Ceilometer needs to know about some of the nova configuration options, so the simplest way to start is copying /etc/nova/nova.conf to /etc/ceilometer-agent.conf. Some of the logging settings used in nova break ceilometer, so they need to be removed. For example, as a user with root permissions:

$ grep -v format_string /etc/nova/nova.conf > /etc/ceilometer-agent.conf

Refer to Configuration Options for details about any other options you might want to modify before starting the service.

6. Start the agent.

$ ./bin/ceilometer-agent

Note: The default development configuration of the agent logs to stderr, so you may want to run this step using a screen session or other tool for maintaining a long-running program in the background.

2.4 Configuration Options

Todo

Document the configuration options.

2.5 Contributing to Ceilometer

2.5.1 Joining the Project

Contributor License Agreement

In order to contribute to the ceilometer project, you need to have signed OpenStack’s contributor’s agreement.

See Also:
LaunchPad Project

Most of the tools used for OpenStack depend on a launchpad.net ID for authentication. After signing up for a launchpad account, join the “openstack” team to have access to the mailing list and receive notifications of important events.

See Also:

- http://launchpad.net
- http://launchpad.net/ceilometer
- http://launchpad.net/~openstack

2.5.2 Project Hosting Details

- **Bug tracker**  http://launchpad.net/ceilometer
- **Mailing list**  http://lists.launchpad.net/openstack (prefix subjects with [metering] for faster responses)
- **Wiki**  http://wiki.openstack.org/EfficientMetering
- **Code Hosting**  https://github.com/stackforge/ceilometer
- **Code Review**  https://review.openstack.org/#/q/status:open+project:stackforge/ceilometer

See Also:

- Joining the Project

2.5.3 Areas to Contribute

**Plugins**

Ceilometer’s architecture is based heavily on the use of plugins to make it easy to extend to collect new sorts of data or store them in different databases.

See Also:

- System Architecture
- Writing Agent Plugins

**Core**

The core parts of ceilometer, not separated into a plugin, are fairly simple but depend on code that is part of nova right now. One project goal is to move the rest of those dependencies out of nova and into openstack-common. Logging and RPC are already done, but the service and manager base classes still need to move.

See Also:

- https://launchpad.net/openstack-nova
- https://launchpad.net/openstack-common
Testing

The pre-release version of ceilometer has extensive unit tests, but has not seen much run-time in real environments. Setting up a copy of ceilometer to monitor a real OpenStack installation or to perform some load testing would be especially helpful.

See Also:

* Installing and Running the Development Version

2.5.4 Working with the Source

Setting up a Development Sandbox

1. Set up a server or virtual machine to run OpenStack using devstack.
2. Clone the ceilometer project to the machine:
   
   ```
   $ cd /opt/stack
   $ git clone https://github.com/stackforge/ceilometer.git
   $ cd ./ceilometer
   ```
3. Once this is done, you need to setup the review process:
   
   ```
   $ git remote add gerrit ssh://<username>@review.stackforge.org:29418/stackforge/ceilometer.git
   ```
4. If you are preparing a patch, create a topic branch and switch to it before making any changes:
   
   ```
   $ git checkout -b TOPIC-BRANCH
   ```

Running the Tests

Ceilometer includes an extensive set of automated unit tests which are run through tox.

1. Install tox:
   
   ```
   $ sudo pip install tox
   ```
2. Install the test dependencies:
   
   ```
   $ sudo pip install -r /opt/stack/ceilometer/tools/test-requires
   ```
3. Run the unit and code-style tests:
   
   ```
   $ cd /opt/stack/ceilometer
   $ tox -e py27,pep8
   ```

See Also:

* tox

Code Reviews

Ceilometer uses the OpenStack review process for all code and developer documentation contributions. Code reviews are managed through gerrit.

See Also:
2.5.5 Writing Agent Plugins

This documentation gives you some clues on how to write a new agent or plugin for Ceilometer a to use if you wish to instrument a functionality which has not yet been covered by an existing one.

An agent runs on each compute node to poll for resource usage. Each metric collected is tagged with the resource ID (such as an instance) and the owner, including tenant and user IDs. The metrics are then reported to the collector via the message bus. More detailed information follows.

Agent

There is currently only one agent defined for Ceilometer which can be found at: ceilometer/agent/ As you will see in the manager.py file, this agent will automatically load any plugin defined in the namespace ceilometer.poll.compute.

Agents are added by implementing a new nova.manager.Manager class, in the same way it was done for the Agent-Manager for the compute agent in the file ceilometer/agent/manager.py.

Plugins

An agent can support multiple plugins to retrieve different information and send them to the collector. As stated above, an agent will automatically activate all plugins of a given class. For example, the compute agent will load all plugins of class ceilometer.poll.compute. This will load, among others, the ceilometer.compute.libvirt.CPUPollster, which is defined in the file ceilometer/compute/libvirt.py as well as the ceilometer.compute.notifications.InstanceNotifications plugin which is defined in the file ceilometer/compute/notifications.py.

We are using these two existing plugins as examples as the first one provides an example of how to interact when you need to retrieve information from an external system (pollster) and the second one is an example of how to forward an existing event notification on the standard OpenStack queue to ceilometer.

Pollster

Pollsters are defined as subclasses of the ceilometer.plugin.PollsterBase meta class as defined in the ceilometer/plugin.py file. Pollsters must implement one method: get_counters(self, manager, context), which returns a sequence of Counter objects as defined in the ceilometer/counter.py file.

In the CPUPollster plugin, the get_counters method is implemented as a loop which, for each instances running on the local host, retrieves the cpu_time from libvirt and send back two Counter objects. The first one, named “cpu”, is of type “cumulative”, meaning that between two polls, its value is not reset, or in other word that the cpu value is always provided as a duration that continuously increases since the creation of the instance. The second one, named “instance”, is of type “delta”, meaning that it’s value is just the volume since the last poll. Here, the instance counter is only used as a way to tell the system that the instance is still running, hence the hard coded value of 1.

Note that the LOG method is only used as a debugging tool and does not participate in the actual metering activity.
Notifications

Notifications are defined as subclass of the `ceilometer.plugin.NotificationBase` meta class as defined in the `ceilometer/plugin.py` file. Notifications must implement two methods:

```python
def get_event_types(self):
    pass

def process_notification(self, message):
    pass
```

In the `InstanceNotifications` plugin, it listens to three events:

- `compute.instance.create.end`
- `compute.instance.exists`
- `compute.instance.delete.start`

using the `get_event_type` method and subsequently the method `process_notification` will be invoked each time such events are happening which generates the appropriate counter objects to be sent to the collector.

Tests

Any new plugin or agent contribution will only be accepted into the project if provided together with unit tests. Those are defined for the compute agent plugins in the directory `tests/compute` and for the agent itself in `test/agent`. Unit tests are run in a continuous integration process for each commit made to the project, thus ensuring as best as possible that a given patch has no side effect to the rest of the project.

Todo

FIXME: could not find a unit test for CPUPollster

2.6 Glossary

agent  Software service running on the OpenStack infrastructure measuring usage and sending the results to the collector.

API server  HTTP REST API service for ceilometer.

ceilometer  From WikiPedia:

> A ceilometer is a device that uses a laser or other light source to determine the height of a cloud base.

collector  Software service running on the OpenStack infrastructure monitoring notifications from other OpenStack components and meter events from the ceilometer agent and recording the results in the database.

data store  Storage system for recording data collected by ceilometer.

non-repudiable  From WikiPedia:

> Non-repudiation refers to a state of affairs where the purported maker of a statement will not be able to successfully challenge the validity of the statement or contract. The term is often seen in

---


a legal setting wherein the authenticity of a signature is being challenged. In such an instance, the authenticity is being “repudiated”.
CHAPTER THREE

INDICES AND TABLES

• genindex
• modindex
• search
CHAPTER
FOUR

TO DO

Todo
Document the configuration options.

(The original entry is located in /home/docs/sites/readthedocs.org/checkouts/readthedocs.org/user_builds/ceilometer/checkouts/latest/doc/source/configuration.rst, line 20.)

Todo
FIXME: could not find a unit test for CPUPollster

(The original entry is located in /home/docs/sites/readthedocs.org/checkouts/readthedocs.org/user_builds/ceilometer/checkouts/latest/doc/source/contributing/plugins.rst, line 106.)

Todo
Replicate the list of meters here.

(The original entry is located in /home/docs/sites/readthedocs.org/checkouts/readthedocs.org/user_builds/ceilometer/checkouts/latest/doc/source/measurements.rst, line 29.)