LIBRADOS (PYTHON)

The `rados` module is a thin Python wrapper for `librados`.

INSTALLATION

To install Python libraries for Ceph, see Getting librados for Python.

GETTING STARTED

You can create your own Ceph client using Python. The following tutorial will show you how to import the Ceph Python module, connect to a Ceph cluster, and perform object operations as a `client.admin` user.

Note: To use the Ceph Python bindings, you must have access to a running Ceph cluster. To set one up quickly, see Getting Started.

First, create a Python source file for your Ceph client. ::

linenos: sudo vim client.py

IMPORT THE MODULE

To use the `rados` module, import it into your source file.

```python
import rados
```

CONFIGURE A CLUSTER HANDLE

Before connecting to the Ceph Storage Cluster, create a cluster handle. By default, the cluster handle assumes a cluster named `ceph` (i.e., the default for deployment tools, and our Getting Started guides too), and a `client.admin` user name. You may change these defaults to suit your needs.

To connect to the Ceph Storage Cluster, your application needs to know where to find the Ceph Monitor. Provide this information to your application by specifying the path to your Ceph configuration file, which contains the location of the initial Ceph monitors.

```python
import rados, sys

#Create Handle Examples.
cluster = rados.Rados(conffile='ceph.conf')
cluster = rados.Rados(conffile=sys.argv[1])
cluster = rados.Rados(conffile = 'ceph.conf', conf = dict (keyring = '/path/to/keyrin
```

Ensure that the `conffile` argument provides the path and file name of your Ceph configuration file. You may use the `sys` module to avoid hard-coding the Ceph configuration path and file name.

Your Python client also requires a client keyring. For this example, we use the `client.admin` key by default. If you would like to specify the keyring when creating the cluster handle, you may use the `conf` argument. Alternatively, you may specify the keyring path in your Ceph configuration file. For example, you may add something like the following line to you Ceph configuration file:

```
keyring = /path/to/ceph.client.admin.keyring
```

For additional details on modifying your configuration via Python, see Configuration.

CONNECT TO THE CLUSTER

Once you have a cluster handle configured, you may connect to the cluster. With a connection to the cluster, you may execute methods that return information about the cluster.

```python
import rados, sys

cluster = rados.Rados(conffile='ceph.conf')
print "\nlibrados version: " + str(cluster.version())
print "Will attempt to connect to: " + str(cluster.conf_get('mon initial members'

cluster.connect()
print "\nCluster ID: " + cluster.get_fsid()

print "\n\nCluster Statistics"
print "=================="
cluster_stats = cluster.get_cluster_stats()

for key, value in cluster_stats.iteritems():
        print key, value
```

By default, Ceph authentication is on. Your application will need to know the location of the keyring. The `python-ceph` module doesn't have the default location, so you need to specify the keyring path. The easiest way to specify the keyring is to add it to the Ceph configuration file. The following Ceph configuration file example uses the `client.admin` keyring you generated with `ceph-deploy`.

```
[global]
# ... elided configuration
keyring=/path/to/keyring/ceph.client.admin.keyring
```

MANAGE POOLS

When connected to the cluster, the Rados API allows you to manage pools. You can list pools, check for the existence of a pool, create a pool and delete a pool.

```python
print "\n\nPool Operations"
print "==============="

print "\nAvailable Pools"
print "----------------"
pools = cluster.list_pools()

for pool in pools:
        print pool

print "\nCreate 'test' Pool"
print "------------------"
cluster.create_pool('test')

print "\nPool named 'test' exists: " + str(cluster.pool_exists('test'))
print "\nVerify 'test' Pool Exists"
print "-------------------------"
pools = cluster.list_pools()

for pool in pools:
        print pool

print "\nDelete 'test' Pool"
print "------------------"
cluster.delete_pool('test')
print "\nPool named 'test' exists: " + str(cluster.pool_exists('test'))
```

INPUT/OUTPUT CONTEXT

Reading from and writing to the Ceph Storage Cluster requires an input/output context (ioctx). You can create an ioctx with the `open_ioctx()` or `open_ioctx2()` method of the Rados class. The `ioctx_name` parameter is the name of the pool and `pool_id` is the ID of the pool you wish to use.

```
ioctx = cluster.open_ioctx('data')
```

or

```
ioctx = cluster.open_ioctx2(pool_id)
```

Once you have an I/O context, you can read/write objects, extended attributes, and perform a number of other operations. After you complete operations, ensure that you close the connection. For example:

```
print "\nClosing the connection."
ioctx.close()
```

WRITING, READING AND REMOVING OBJECTS

Once you create an I/O context, you can write objects to the cluster. If you write to an object that doesn’t exist, Ceph creates it. If you write to an object that exists, Ceph overwrites it (except when you specify a range, and then it only overwrites the range). You may read objects (and object ranges) from the cluster. You may also remove objects from the cluster. For example:

```
print "\nWriting object 'hw' with contents 'Hello World!' to pool 'data'."
ioctx.write_full("hw", "Hello World!")

print "\n\nContents of object 'hw'\n------------------------\n"
print ioctx.read("hw")

print "\nRemoving object 'hw'"
ioctx.remove_object("hw")
```

WRITING AND READING XATTRS

Once you create an object, you can write extended attributes (XATTRs) to the object and read XATTRs from the object. For example:

```
print "\n\nWriting XATTR 'lang' with value 'en_US' to object 'hw'"
ioctx.set_xattr("hw", "lang", "en_US")

print "\n\nGetting XATTR 'lang' from object 'hw'\n"
print ioctx.get_xattr("hw", "lang")
```

LISTING OBJECTS

If you want to examine the list of objects in a pool, you may retrieve the list of objects and iterate over them with the object iterator. For example:

```
object_iterator = ioctx.list_objects()

while True :

        try :
                rados_object = object_iterator.next()
                print "Object contents = " + rados_object.read()

        except StopIteration :
                break
```

The `Object` class provides a file-like interface to an object, allowing you to read and write content and extended attributes. Object operations using the I/O context provide additional functionality and asynchronous capabilities.

CLUSTER HANDLE API

The `Rados` class provides an interface into the Ceph Storage Daemon.

CONFIGURATION

The `Rados` class provides methods for getting and setting configuration values, reading the Ceph configuration file, and parsing arguments. You do not need to be connected to the Ceph Storage Cluster to invoke the following methods. See Storage Cluster Configuration for details on settings.

`Rados.`**`conf_get`**(*option*)

`Rados.`**`conf_set`**(*option*, *val*)

`Rados.`**`conf_read_file`**(*path=None*)

`Rados.`**`conf_parse_argv`**(*args*)

Rados.conf_parse_argv(self, args)

Parse known arguments from args, and remove; returned args contain only those unknown to ceph

`Rados.`**`version`**()

Rados.version(self)

Get the version number of the `librados` C library.

Returns: a tuple of `(major, minor, extra)` components of the librados version

CONNECTION MANAGEMENT

Once you configure your cluster handle, you may connect to the cluster, check the cluster `fsid`, retrieve cluster statistics, and disconnect (shutdown) from the cluster. You may also assert that the cluster handle is in a particular state (e.g., “configuring”, “connecting”, etc.).

`Rados.`**`connect`**(*timeout=0*)

Rados.connect(self, timeout=0)

Connect to the cluster. Use shutdown() to release resources.

`Rados.`**`shutdown`**()

Rados.shutdown(self)

Disconnects from the cluster. Call this explicitly when a Rados.connect()ed object is no longer used.

`Rados.`**`get_fsid`**()

Rados.get_fsid(self)

Get the fsid of the cluster as a hexadecimal string.

Raises: **`Error`**
Returns: str - cluster fsid

`Rados.`**`get_cluster_stats`**()

Rados.get_cluster_stats(self)

Read usage info about the cluster

This tells you total space, space used, space available, and number of objects. These are not updated immediately when data is written, they are eventually consistent.

Returns: dict - contains the following keys:

- `kb` (int) - total space
- `kb_used` (int) - space used
- `kb_avail` (int) - free space available
- `num_objects` (int) - number of objects

class `rados.`**`Rados`**

`require_state`(**args*)

Checks if the Rados object is in a special state

Parameters: **args** – Any number of states to check as separate arguments

Raises: **`RadosStateError`**

POOL OPERATIONS

To use pool operation methods, you must connect to the Ceph Storage Cluster first. You may list the available pools, create a pool, check to see if a pool exists, and delete a pool.

`Rados.`**`list_pools`**()

Rados.list_pools(self)

Gets a list of pool names.

Returns: list - of pool names.

`Rados.`**`create_pool`**(*pool_name*, *auid=None*, *crush_rule=None*)

`Rados.`**`pool_exists`**()

`Rados.`**`delete_pool`**(*pool_name*)

INPUT/OUTPUT CONTEXT API

To write data to and read data from the Ceph Object Store, you must create an Input/Output context (ioctx). The *Rados* class provides *open_ioctx()* and *open_ioctx2()* methods. The remaining `ioctx` operations involve invoking methods of the *Ioctx* and other classes.

`Rados.`**`open_ioctx`**(*ioctx_name*)

`Ioctx.`**`require_ioctx_open`**()

Ioctx.require_ioctx_open(self)

Checks if the rados.Ioctx object state is ‘open’

Raises: IoctxStateError

`Ioctx.`**`get_stats`**()

Ioctx.get_stats(self)

Get pool usage statistics

Returns: dict - contains the following keys:

- `num_bytes` (int) - size of pool in bytes
- `num_kb` (int) - size of pool in kbytes
- `num_objects` (int) - number of objects in the pool
- `num_object_clones` (int) - number of object clones
- `num_object_copies` (int) - number of object copies
- `num_objects_missing_on_primary` (int) - number of objets

  missing on primary

- `num_objects_unfound` (int) - number of unfound objects
- `num_objects_degraded` (int) - number of degraded objects
- `num_rd` (int) - bytes read
- `num_rd_kb` (int) - kbytes read
- `num_wr` (int) - bytes written
- `num_wr_kb` (int) - kbytes written

`Ioctx.`**`change_auid`**(*auid*)

Ioctx.change_auid(self, auid)

Attempt to change an io context's associated auid "owner."

Requires that you have write permission on both the current and new auid.

Raises: **`Error`**

`Ioctx.`**`get_last_version`**()

Ioctx.get_last_version(self)

Return the version of the last object read or written to.

This exposes the internal version number of the last object read or written via this io context

Returns: version of the last object used

`Ioctx.`**`close`**()

Ioctx.close(self)

Close a rados.Ioctx object.

This just tells librados that you no longer need to use the io context. It may not be freed immediately if there are pending asynchronous requests on it, but you should not use an io context again after calling this function on it.

OBJECT OPERATIONS

The Ceph Storage Cluster stores data as objects. You can read and write objects synchronously or asynchronously. You can read and write from offsets. An object has a name (or key) and data.

`Ioctx.`**`aio_write`**(*object_name*, *to_write*, *offset=0*, *oncomplete=None*, *onsafe=None*)

`Ioctx.`**`aio_write_full`**(*object_name*, *to_write*, *oncomplete=None*, *onsafe=None*)

`Ioctx.`**`aio_append`**(*object_name*, *to_append*, *oncomplete=None*, *onsafe=None*)

`Ioctx.`**`write`**(*key*, *data*, *offset=0*)

`Ioctx.`**`write_full`**(*key*, *data*)

`Ioctx.`**`aio_flush`**()

Ioctx.aio_flush(self)

Block until all pending writes in an io context are safe

Raises: **`Error`**

`Ioctx.`**`set_locator_key`**(*loc_key*)

`Ioctx.`**`aio_read`**(*object_name*, *length*, *offset*, *oncomplete*)

`Ioctx.`**`read`**(*key*, *length=8192*, *offset=0*)

`Ioctx.`**`stat`**(*key*)

`Ioctx.`**`trunc`**(*key*, *size*)

`Ioctx.`**`remove_object`**(*key*)

OBJECT EXTENDED ATTRIBUTES

You may set extended attributes (XATTRs) on an object. You can retrieve a list of objects or XATTRs and iterate over them.

`Ioctx.`**`set_xattr`**(*key*, *xattr_name*, *xattr_value*)

`Ioctx.`**`get_xattrs`**(*oid*)

`XattrIterator.`**`__next__`**()

Get the next xattr on the object

Raises: StopIteration
Returns: pair - of name and value of the next Xattr

`Ioctx.`**`get_xattr`**(*key*, *xattr_name*)

`Ioctx.`**`rm_xattr`**(*key*, *xattr_name*)

OBJECT INTERFACE

From an I/O context, you can retrieve a list of objects from a pool and iterate over them. The object interface provide makes each object look like a file, and you may perform synchronous operations on the objects. For asynchronous operations, you should use the I/O context methods.

`Ioctx.`**`list_objects`**()

Ioctx.list_objects(self)

Get ObjectIterator on rados.Ioctx object.

Returns: ObjectIterator

`ObjectIterator.`**`__next__`**()

Get the next object name and locator in the pool

Raises: StopIteration
Returns: next rados.Ioctx Object

`Object.`**`read`**(*length = 1024*1024*)

`Object.`**`write`**(*string_to_write*)

`Object.`**`get_xattrs`**()

`Object.`**`get_xattr`**(*xattr_name*)

`Object.`**`set_xattr`**(*xattr_name*, *xattr_value*)

`Object.`**`rm_xattr`**(*xattr_name*)

`Object.`**`stat`**()

`Object.`**`remove`**()