LIBRADOS (C)

librados provides low-level access to the RADOS service. For an overview of RADOS, see Architecture.

EXAMPLE: CONNECTING AND WRITING AN OBJECT

To use *Librados*, you instantiate a **rados_t** variable (a cluster handle) and call **rados_create()** with a pointer to it:

```
int err;
rados_t cluster;

err = rados_create(&cluster, NULL);
if (err < 0) {
        fprintf(stderr, "%s: cannot create a cluster handle: %s\n", argv[0], strerror(-err));
        exit(1);
}
```

Then you configure your **rados_t** to connect to your cluster, either by setting individual values (**rados_conf_set()**), using a configuration file (**rados_conf_read_file()**), using command line options (**rados_conf_parse_argv()**), or an environment variable (**rados_conf_parse_env()**):

```
err = rados_conf_read_file(cluster, "/path/to/myceph.conf");
if (err < 0) {
        fprintf(stderr, "%s: cannot read config file: %s\n", argv[0], strerror(-err));
        exit(1);
}
```

Once the cluster handle is configured, you can connect to the cluster with **rados_connect()**:

```
err = rados_connect(cluster);
if (err < 0) {
        fprintf(stderr, "%s: cannot connect to cluster: %s\n", argv[0], strerror(-err));
        exit(1);
}
```

Then you open an "IO context", a **rados_ioctx_t**, with **rados_ioctx_create()**:

```
rados_ioctx_t io;
char *poolname = "mypool";

err = rados_ioctx_create(cluster, poolname, &io);
if (err < 0) {
        fprintf(stderr, "%s: cannot open rados pool %s: %s\n", argv[0], poolname, strerror(-e
        rados_shutdown(cluster);
        exit(1);
}
```

Note that the pool you try to access must exist.

Then you can use the RADOS data manipulation functions, for example write into an object called `greeting` with **rados_write_full()**:

```
err = rados_write_full(io, "greeting", "hello", 5);
if (err < 0) {
        fprintf(stderr, "%s: cannot write pool %s: %s\n", argv[0], poolname, strerror(-err));
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
}
```

In the end, you'll want to close your IO context and connection to RADOS with `rados_ioctx_destroy()` and `rados_shutdown()`:

```
rados_ioctx_destroy(io);
rados_shutdown(cluster);
```

ASYCHRONOUS IO

When doing lots of IO, you often don't need to wait for one operation to complete before starting the next one. *Librados* provides asynchronous versions of several operations:

- `rados_aio_write()`
- `rados_aio_append()`
- `rados_aio_write_full()`
- `rados_aio_read()`

For each operation, you must first create a `rados_completion_t` that represents what to do when the operation is safe or complete by calling `rados_aio_create_completion()`. If you don't need anything special to happen, you can pass NULL:

```
rados_completion_t comp;
err = rados_aio_create_completion(NULL, NULL, NULL, &comp);
if (err < 0) {
        fprintf(stderr, "%s: could not create aio completion: %s\n", argv[0], strerror(-err))
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
}
```

Now you can call any of the aio operations, and wait for it to be in memory or on disk on all replicas:

```
err = rados_aio_write(io, "foo", comp, "bar", 3, 0);
if (err < 0) {
        fprintf(stderr, "%s: could not schedule aio write: %s\n", argv[0], strerror(-err));
        rados_aio_release(comp);
        rados_ioctx_destroy(io);
        rados_shutdown(cluster);
        exit(1);
}
rados_wait_for_complete(comp); // in memory
rados_wait_for_safe(comp); // on disk
```

Finally, we need to free the memory used by the completion with `rados_aio_release()`:

```
rados_aio_release(comp);
```

You can use the callbacks to tell your application when writes are durable, or when read buffers are full. For example, if you wanted to measure the latency of each operation when appending to several objects, you could schedule several writes and store the ack and commit time in the corresponding callback, then wait for all of them to complete using `rados_aio_flush()` before analyzing the latencies:

```
typedef struct {
        struct timeval start;
        struct timeval ack_end;
        struct timeval commit_end;
} req_duration;

void ack_callback(rados_completion_t comp, void *arg) {
        req_duration *dur = (req_duration *) arg;
        gettimeofday(&dur->ack_end, NULL);
}

void commit_callback(rados_completion_t comp, void *arg) {
```

```
        req_duration *dur = (req_duration *) arg;
        gettimeofday(&dur->commit_end, NULL);
}

int output_append_latency(rados_ioctx_t io, const char *data, size_t len, size_t num_writes)
        req_duration times[num_writes];
        rados_completion_t comps[num_writes];
        for (size_t i = 0; i < num_writes; ++i) {
                gettimeofday(&times[i].start, NULL);
                int err = rados_aio_create_completion((void*) &times[i], ack_callback, commit
                if (err < 0) {
                        fprintf(stderr, "Error creating rados completion: %s\n", strerror(-er
                        return err;
                }
                char obj_name[100];
                snprintf(obj_name, sizeof(obj_name), "foo%ld", (unsigned long)i);
                err = rados_aio_append(io, obj_name, comps[i], data, len);
                if (err < 0) {
                        fprintf(stderr, "Error from rados_aio_append: %s", strerror(-err));
                        return err;
                }
        }
        // wait until all requests finish *and* the callbacks complete
        rados_aio_flush(io);
        // the latencies can now be analyzed
        printf("Request # | Ack latency (s) | Commit latency (s)\n");
        for (size_t i = 0; i < num_writes; ++i) {
                // don't forget to free the completions
                rados_aio_release(comps[i]);
                struct timeval ack_lat, commit_lat;
                timersub(&times[i].ack_end, &times[i].start, &ack_lat);
                timersub(&times[i].commit_end, &times[i].start, &commit_lat);
                printf("%9ld | %8ld.%06ld | %10ld.%06ld\n", (unsigned long) i, ack_lat.tv_sec
        }
        return 0;
}
```

Note that all the `rados_completion_t` must be freed with `rados_aio_release()` to avoid leaking memory.

API CALLS

STRUCT RADOS_POOL_STAT_T

struct **rados_pool_stat_t**

Usage information for a pool.

MEMBERS

uint64_t **num_bytes**

space used in bytes

uint64_t **num_kb**

space used in KB

uint64_t **num_objects**

number of objects in the pool

uint64_t **num_object_clones**

number of clones of objects

uint64_t **num_object_copies**

num_objects * num_replicas

uint64_t **num_objects_missing_on_primary**

uint64_t **num_objects_unfound**

number of objects found on no OSDs

uint64_t **num_objects_degraded**
number of objects replicated fewer times than they should be (but found on at least one **OSD**)

uint64_t **num_rd**

uint64_t **num_rd_kb**

uint64_t **num_wr**

uint64_t **num_wr_kb**

STRUCT RADOS_CLUSTER_STAT_T

struct **rados_cluster_stat_t**
Cluster-wide usage information.

MEMBERS

uint64_t **kb**

uint64_t **kb_used**

uint64_t **kb_avail**

uint64_t **num_objects**

DEFINES

CEPH_OSD_TMAP_HDR

CEPH_OSD_TMAP_SET

CEPH_OSD_TMAP_CREATE

CEPH_OSD_TMAP_RM

LIBRADOS_VER_MAJOR

LIBRADOS_VER_MINOR

LIBRADOS_VER_EXTRA

LIBRADOS_VERSION

LIBRADOS_VERSION_CODE

LIBRADOS_SUPPORTS_WATCH

LIBRADOS_SNAP_HEAD

LIBRADOS_SNAP_DIR

TYPES

rados_t
A handle for interacting with a RADOS cluster.

It encapsulates all RADOS client configuration, including username, key for authentication, logging, and debugging. Talking different clusters

Warning: asphyxiate: No renderer found for doxygen tag ‘ndash’

```
<ndash xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/> or to the same c
```

or to the same cluster with different users

Warning: asphyxiate: No renderer found for doxygen tag ‘ndash’

```
<ndash xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/> requires differe
```

requires different cluster handles.

rados_config_t

rados_config_t

A handle for the ceph configuration context for the rados_t cluster instance. This can be used to share configuration context/state (e.g., logging configuration) between librados instance.

Warning: The config context does not have independent reference counting. As such, a rados_config_t handle retrieved from a given rados_t is only valid as long as that rados_t.

rados_ioctx_t

An io context encapsulates a few settings for all I/O operations done on it:

- pool - set when the io context is created (see **rados_ioctx_create()**)
- snapshot context for writes (see **rados_ioctx_selfmanaged_snap_set_write_ctx()**)
- snapshot id to read from (see **rados_ioctx_snap_set_read()**)
- object locator for all single-object operations (see **rados_ioctx_locator_set_key()**)

Warning: changing any of these settings is not thread-safe - librados users must synchronize any of these changes on their own, or use separate io contexts for each thread

rados_list_ctx_t

An iterator for listing the objects in a pool.

Used with **rados_objects_list_open()**,**rados_objects_list_next()**, and**rados_objects_list_close()**.

rados_snap_t

The id of a snapshot.

rados_xattrs_iter_t

An iterator for listing extended attrbutes on an object.

Used with **rados_getxattrs()**,**rados_getxattrs_next()**, and**rados_getxattrs_end()**.

rados_completion_t

Represents the state of an asynchronous operation - it contains the return value once the operation completes, and can be used to block until the operation is complete or safe.

rados_callback_t

Callbacks for asynchrous operations take two parameters:

- cb the completion that has finished
- arg application defined data made available to the callback function

rados_watchcb_t

Callback activated when a notify is received on a watched object.

Parameters are:

- opcode undefined
- ver version of the watched object
- arg application-specific data

Note: BUG: opcode is an internal detail that shouldn’t be exposed

FUNCTIONS

void **rados_version**(int *major, int *minor, int *extra)

Get the version of librados.

The version number is major.minor.extra. Note that this is unrelated to the Ceph version number.

TODO: define version semantics, i.e.:

- incrementing major is for backwards-incompatible changes
- incrementing minor is for backwards-compatible changes
- incrementing extra is for bug fixes

Parameters:
- **major** – where to store the major version number
- **minor** – where to store the minor version number
- **extra** – where to store the extra version number

int **rados_create**(rados_t *cluster, const char *const id)

Create a handle for communicating with a RADOS cluster.

Ceph environment variables are read when this is called, so if $CEPH_ARGS specifies everything you need to connect, no further configuration is necessary.

Parameters:
- **cluster** – where to store the handle
- **id** – the user to connect as (i.e. admin, not client.admin)

Returns: 0 on success, negative error code on failure

int **rados_create_with_context**(rados_t *cluster, rados_config_t cct)

Initialize a cluster handle from an existing configuration.

Share configuration state with another rados_t instance.

Parameters:
- **cluster** – where to store the handle
- **cct_** – the existing configuration to use

Returns: 0 on success, negative error code on failure

int **rados_connect**(rados_t cluster)

Connect to the cluster.

Note: BUG: Before calling this, calling a function that communicates with the cluster will crash.

Precondition: The cluster handle is configured with at least a monitor address. If cephx is enabled, a client name and secret must also be set.

Postcondition: If this succeeds, any function in librados may be used

Parameters:
- **cluster** – The cluster to connect to.

Returns: 0 on sucess, negative error code on failure

void **rados_shutdown**(rados_t cluster)

Disconnects from the cluster.

For clean up, this is only necessary after `rados_connect()` has succeeded.

Warning: This does not guarantee any asynchronous writes have completed. To do that, you must call on all open io contexts.

Postcondition: the cluster handle cannot be used again

Parameters:
- **cluster** – the cluster to shutdown

int **rados_conf_read_file**(rados_t cluster, const char *path)

Configure the cluster handle using a Ceph config file.

If path is NULL, the default locations are searched, and the first found is used. The locations are:

- $CEPH_CONF (environment variable)
- /etc/ceph/ceph.conf
- ~/.ceph/config
- ceph.conf (in the current working directory)

Precondition: `rados_connect()` has not been called on the cluster handle

Parameters:
- **cluster** – cluster handle to configure
- **path** – path to a Ceph configuration file

Returns: 0 on success, negative error code on failure

int **rados_conf_parse_argv**(rados_t *cluster*, int *argc*, const char *******argv*)

Configure the cluster handle with command line arguments.

argv can contain any common Ceph command line option, including any configuration parameter prefixed by ‘

Warning: asphyxiate: No renderer found for doxygen tag ‘ndash’

```
<ndash xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>' and replacing s
```

‘ and replacing spaces with dashes or underscores. For example, the following options are equivalent:

- **Warning:** asphyxiate: No renderer found for doxygen tag ‘ndash’

  ```
  <ndash xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>mon-host 10.0
  ```

  mon-host 10.0.0.1:6789
- **Warning:** asphyxiate: No renderer found for doxygen tag ‘ndash’

  ```
  <ndash xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>mon_host 10.0
  ```

  mon_host 10.0.0.1:6789
- -m 10.0.0.1:6789

Precondition: `rados_connect()` has not been called on the cluster handle

Parameters:
- **cluster** – cluster handle to configure
- **argc** – number of arguments in argv
- **argv** – arguments to parse

Returns: 0 on success, negative error code on failure

int **rados_conf_parse_env**(rados_t *cluster*, const char ******var*)

Configure the cluster handle based on an environment variable.

The contents of the environment variable are parsed as if they were Ceph command line options. If var is NULL, the CEPH_ARGS environment variable is used.

Precondition: `rados_connect()` has not been called on the cluster handle

Note: BUG: this is not threadsafe - it uses a static buffer

Parameters:
- **cluster** – cluster handle to configure
- **var** – name of the environment variable to read

Returns: 0 on success, negative error code on failure

int **rados_conf_set**(rados_t *cluster*, const char **option*, const char **value*)

Set a configuration option.

Precondition: `rados_connect()` has not been called on the cluster handle

Parameters:
- **cluster** – cluster handle to configure
- **option** – option to set
- **value** – value of the option

Returns: 0 on success, negative error code on failure
-ENOENT when the option is not a Ceph configuration option

int **rados_conf_get**(rados_t *cluster*, const char **option*, char **buf*, size_t *len*)

Get the value of a configuration option.

Parameters:
- **cluster** – configuration to read
- **option** – which option to read
- **buf** – where to write the configuration value
- **len** – the size of buf in bytes

Returns: 0 on success, negative error code on failure
-ENAMETOOLONG if the buffer is too short to contain the requested value

int **rados_cluster_stat**(rados_t *cluster*, struct rados_cluster_stat_t **result*)

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.

Parameters:
- **cluster** – cluster to query
- **result** – where to store the results

Returns: 0 on success, negative error code on failure

int **rados_cluster_fsid**(rados_t *cluster*, char **buf*, size_t *len*)

Get the fsid of the cluster as a hexadecimal string.

The fsid is a unique id of an entire Ceph cluster.

Parameters:
- **cluster** – where to get the fsid
- **buf** – where to write the fsid
- **len** – the size of buf in bytes (should be 37)

Returns: 0 on success, negative error code on failure
-ERANGE if the buffer is too short to contain the fsid

int **rados_pool_list**(rados_t *cluster*, char **buf*, size_t *len*)

List pools.

Gets a list of pool names as NULL-terminated strings. The pool names will be placed in the supplied buffer one after another. After the last pool name, there will be two 0 bytes in a row.

If len is too short to fit all the pool name entries we need, we will fill as much as we can.

Parameters:
- **cluster** – cluster handle
- **buf** – output buffer
- **len** – output buffer length

Returns: length of the buffer we would need to list all pools

rados_config_t **rados_cct**(rados_t *cluster*)

Get a configuration handle for a rados cluster handle.

This handle is valid only as long as the cluster handle is valid.

Parameters:
- **cluster** – cluster handle

Returns: config handle for this cluster

uint64_t **rados_get_instance_id**(rados_t *cluster*)

Get a global id for current instance.

This id is a unique representation of current connection to the cluster

Parameters:
- **cluster** – cluster handle

Returns: instance global id

int **rados_ioctx_create**(rados_t *cluster*, const char **pool_name*, rados_ioctx_t **ioctx*)

Create an io context.

The io context allows you to perform operations within a particular pool. For more details see rados_ioctx_t.

Parameters:
- **cluster** – which cluster the pool is in
- **pool_name** – name of the pool
- **ioctx** – where to store the io context

Returns: 0 on success, negative error code on failure

void **rados_ioctx_destroy**(rados_ioctx_t *io*)

The opposite of rados_ioctx_create.

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.

Warning: This does not guarantee any asynchronous writes have completed. You must call on the io context before destroying it to do that.

Parameters:
- **io** – the io context to dispose of

rados_config_t **rados_ioctx_cct**(rados_ioctx_t *io*)

Get configuration hadnle for a pool handle.

Parameters:
- **io** – pool handle

Returns: rados_config_t for this cluster

rados_t **rados_ioctx_get_cluster**(rados_ioctx_t *io*)

Get the cluster handle used by this rados_ioctx_t Note that this is a weak reference, and should not be destroyed via rados_destroy().

Parameters:
- **io** – the io context

Returns: the cluster handle for this io context

int **rados_ioctx_pool_stat**(rados_ioctx_t *io*, struct rados_pool_stat_t **stats*)

Get pool usage statistics.

Fills in a **rados_pool_stat_t**after querying the cluster.

Parameters:
- **io** – determines which pool to query
- **stats** – where to store the results

Returns: 0 on success, negative error code on failure

int64_t **rados_pool_lookup**(rados_t *cluster*, const char **pool_name*)

Get the id of a pool.

Parameters:
- **cluster** – which cluster the pool is in
- **pool_name** – which pool to look up

Returns: id of the pool
-ENOENT if the pool is not found

int **rados_pool_reverse_lookup**(rados_t *cluster*, int64_t *id*, char **buf*, size_t *maxlen*)

Get the name of a pool.

Parameters:
- **cluster** – which cluster the pool is in
- **id** – the id of the pool
- **buf** – where to store the pool name
- **maxlen** – size of buffer where name will be stored

Returns: length of string stored, or -ERANGE if buffer too small

int **rados_pool_create**(rados_t *cluster*, const char **pool_name*)

Create a pool with default settings.

The default owner is the admin user (auid 0). The default crush rule is rule 0.

Parameters:
- **cluster** – the cluster in which the pool will be created
- **pool_name** – the name of the new pool

Returns: 0 on success, negative error code on failure

int **rados_pool_create_with_auid**(rados_t *cluster*, const char **pool_name*, uint64_t *auid*)

Create a pool owned by a specific auid.

The auid is the authenticated user id to give ownership of the pool. TODO: document auid and the rest of the auth system

Parameters:
- **cluster** – the cluster in which the pool will be created
- **pool_name** – the name of the new pool
- **auid** – the id of the owner of the new pool

Returns: 0 on success, negative error code on failure

int **rados_pool_create_with_crush_rule**(rados_t *cluster*, const char **pool_name*, __u8 *crush_rule_num*)

Create a pool with a specific CRUSH rule.

Parameters:
- **cluster** – the cluster in which the pool will be created
- **pool_name** – the name of the new pool
- **crush_rule_num** – which rule to use for placement in the new pool1

Returns: 0 on success, negative error code on failure

int **rados_pool_create_with_all**(rados_t *cluster*, const char **pool_name*, uint64_t *auid*, __u8 *crush_rule_num*)

Create a pool with a specific CRUSH rule and auid.

This is a combination of **rados_pool_create_with_crush_rule()**and**rados_pool_create_with_auid()**.

Parameters:
- **cluster** – the cluster in which the pool will be created
- **pool_name** – the name of the new pool
- **crush_rule_num** – which rule to use for placement in the new pool2
- **auid** – the id of the owner of the new pool

Returns: 0 on success, negative error code on failure

int **rados_pool_delete**(rados_t *cluster*, const char **pool_name*)

Delete a pool and all data inside it.

The pool is removed from the cluster immediately, but the actual data is deleted in the background.

Parameters:
- **cluster** – the cluster the pool is in
- **pool_name** – which pool to delete

Returns: 0 on success, negative error code on failure

int **rados_ioctx_pool_set_auid**(rados_ioctx_t *io*, uint64_t *auid*)

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

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

Parameters:
- **io** – reference to the pool to change.
- **auid** – the auid you wish the io to have.

Returns: 0 on success, negative error code on failure

int **rados_ioctx_pool_get_auid**(rados_ioctx_t *io*, uint64_t **auid*)

Get the auid of a pool.

Parameters:
- **io** – pool to query
- **auid** – where to store the auid

Returns: 0 on success, negative error code on failure

int64_t **rados_ioctx_get_id**(rados_ioctx_t *io*)

Get the pool id of the io context.

Parameters:
- **io** – the io context to query

Returns: the id of the pool the io context uses

int **rados_ioctx_get_pool_name**(rados_ioctx_t *io*, char **buf*, unsigned *maxlen*)

Get the pool name of the io context.

Parameters:
- **io** – the io context to query
- **buf** – pointer to buffer where name will be stored
- **maxlen** – size of buffer where name will be stored

Returns: length of string stored, or -ERANGE if buffer too small

void **rados_ioctx_locator_set_key**(rados_ioctx_t *io*, const char **key*)

Set the key for mapping objects to pgs within an io context.

The key is used instead of the object name to determine which placement groups an object is put in. This affects all subsequent operations of the io context - until a different locator key is set, all objects in this io context will be placed in the same pg.

This is useful if you need to do clone_range operations, which must be done with the source and destination objects in the same pg.

Parameters:
- **io** – the io context to change
- **key** – the key to use as the object locator, or NULL to discard any previously set key

int **rados_objects_list_open**(rados_ioctx_t *io*, rados_list_ctx_t **ctx*)

Start listing objects in a pool.

Parameters:
- **io** – the pool to list from
- **ctx** – the handle to store list context in

Returns: 0 on success, negative error code on failure

int **rados_objects_list_next**(rados_list_ctx_t *ctx*, const char ***entry*, const char ***key*)

Get the next object name and locator in the pool.

Warning: asphyxiate: No renderer found for doxygen tag ‘emphasis’

```
<emphasis xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">entry and *key
```

Parameters:
- **ctx** – iterator marking where you are in the listing
- **entry** – where to store the name of the entry
- **key** – where to store the object locator (set to NULL to ignore)

Returns: 0 on success, negative error code on failure
-ENOENT when there are no more objects to list

void **rados_objects_list_close**(rados_list_ctx_t *ctx*)

Close the object listing handle.

This should be called when the handle is no longer needed. The handle should not be used after it has been

closed.

Parameters:
- **ctx** – the handle to close

int **rados_ioctx_snap_create**(rados_ioctx_t *io*, const char **snapname*)

Create a pool-wide snapshot.

Parameters:
- **io** – the pool to snapshot
- **snapname** – the name of the snapshot

Returns: 0 on success, negative error code on failure

int **rados_ioctx_snap_remove**(rados_ioctx_t *io*, const char **snapname*)

Delete a pool snapshot.

Parameters:
- **io** – the pool to delete the snapshot from
- **snapname** – which snapshot to delete

Returns: 0 on success, negative error code on failure

int **rados_rollback**(rados_ioctx_t *io*, const char **oid*, const char **snapname*)

Rollback an object to a pool snapshot.

The contents of the object will be the same as when the snapshot was taken.

Parameters:
- **io** – the pool in which the object is stored
- **oid** – the name of the object to rollback
- **snapname** – which snapshot to rollback to

Returns: 0 on success, negative error code on failure

void **rados_ioctx_snap_set_read**(rados_ioctx_t *io*, rados_snap_t *snap*)

Set the snapshot from which reads are performed.

Subsequent reads will return data as it was at the time of that snapshot.

Parameters:
- **io** – the io context to change
- **snap** – the id of the snapshot to set, or LIBRADOS_SNAP_HEAD for no snapshot (i.e. normal operation)

int **rados_ioctx_selfmanaged_snap_create**(rados_ioctx_t *io*, rados_snap_t **snapid*)

Allocate an ID for a self-managed snapshot.

Get a unique ID to put in the snaphot context to create a snapshot. A clone of an object is not created until a write with the new snapshot context is completed.

Parameters:
- **io** – the pool in which the snapshot will exist
- **snapid** – where to store the newly allocated snapshot ID

Returns: 0 on success, negative error code on failure

int **rados_ioctx_selfmanaged_snap_remove**(rados_ioctx_t *io*, rados_snap_t *snapid*)

Remove a self-managed snapshot.

This increases the snapshot sequence number, which will cause snapshots to be removed lazily.

Parameters:
- **io** – the pool in which the snapshot will exist
- **snapid** – where to store the newly allocated snapshot ID

Returns: 0 on success, negative error code on failure

int **rados_ioctx_selfmanaged_snap_rollback**(rados_ioctx_t *io*, const char **oid*, rados_snap_t *snapid*)

Rollback an object to a self-managed snapshot.

The contents of the object will be the same as when the snapshot was taken.

Parameters:
- **io** – the pool in which the object is stored
- **oid** – the name of the object to rollback
- **snapid** – which snapshot to rollback to

Returns: 0 on success, negative error code on failure

int **rados_ioctx_selfmanaged_snap_set_write_ctx**(rados_ioctx_t *io*, rados_snap_t *seq*, rados_snap_t **snaps*, int *num_snaps*)

Set the snapshot context for use when writing to objects.

This is stored in the io context, and applies to all future writes.

Parameters:
- **io** – the io context to change
- **seq** – the newest snapshot sequence number for the pool
- **snaps** – array of snapshots in sorted by descending id
- **num_snaps** – how many snaphosts are in the snaps array

Returns: 0 on success, negative error code on failure
-EINVAL if snaps are not in descending order

int **rados_ioctx_snap_list**(rados_ioctx_t *io*, rados_snap_t **snaps*, int *maxlen*)

List all the ids of pool snapshots.

If the output array does not have enough space to fit all the snapshots, -ERANGE is returned and the caller should retry with a larger array.

Parameters:
- **io** – the pool to read from
- **snaps** – where to store the results
- **maxlen** – the number of rados_snap_t that fit in the snaps array

Returns: number of snapshots on success, negative error code on failure
-ERANGE is returned if the snaps array is too short

int **rados_ioctx_snap_lookup**(rados_ioctx_t *io*, const char **name*, rados_snap_t **id*)

Get the id of a pool snapshot.

Parameters:
- **io** – the pool to read from
- **name** – the snapshot to find
- **id** – where to store the result

Returns: 0 on success, negative error code on failure

int **rados_ioctx_snap_get_name**(rados_ioctx_t *io*, rados_snap_t *id*, char **name*, int *maxlen*)

Get the name of a pool snapshot.

Parameters:
- **io** – the pool to read from
- **id** – the snapshot to find
- **name** – where to store the result
- **maxlen** – the size of the name array

Returns: 0 on success, negative error code on failure
-ERANGE if the name array is too small

int **rados_ioctx_snap_get_stamp**(rados_ioctx_t *io*, rados_snap_t *id*, time_t **t*)

Find when a pool snapshot occurred.

Parameters:
- **io** – the pool the snapshot was taken in
- **id** – the snapshot to lookup
- **t** – where to store the result

Returns: 0 on success, negative error code on failure

uint64_t **rados_get_last_version**(rados_ioctx_t *io*)

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

Parameters:
- **io** – the io context to check

Returns: last read or written object version

int **rados_write**(rados_ioctx_t *io*, const char **oid*, const char **buf*, size_t *len*, uint64_t *off*)

Write data to an object.

Parameters:
- **io** – the io context in which the write will occur
- **oid** – name of the object
- **buf** – data to write
- **len** – length of the data, in bytes
- **off** – byte offset in the object to begin writing at

Returns: number of bytes written on success, negative error code on failure

int **rados_write_full**(rados_ioctx_t *io*, const char **oid*, const char **buf*, size_t *len*)

Write an entire object.

The object is filled with the provided data. If the object exists, it is atomically truncated and then written.

Parameters:
- **io** – the io context in which the write will occur
- **oid** – name of the object
- **buf** – data to write
- **len** – length of the data, in bytes

Returns: 0 on success, negative error code on failure

int **rados_clone_range**(rados_ioctx_t *io*, const char **dst*, uint64_t *dst_off*, const char **src*, uint64_t *src_off*, size_t *len*)

Efficiently copy a portion of one object to another.

If the underlying filesystem on the **OSD**supports it, this will be a copy-on-write clone.

The src and dest objects must be in the same pg. To ensure this, the io context should have a locator key set (see **rados_ioctx_locator_set_key()**).

Parameters:
- **io** – the context in which the data is cloned
- **dst** – the name of the destination object
- **dst_off** – the offset within the destination object (in bytes)
- **src** – the name of the source object
- **src_off** – the offset within the source object (in bytes)
- **len** – how much data to copy

Returns: 0 on success, negative error code on failure

int **rados_append**(rados_ioctx_t *io*, const char **oid*, const char **buf*, size_t *len*)

Append data to an object.

Parameters:
- **io** – the context to operate in
- **oid** – the name of the object
- **buf** – the data to append
- **len** – length of buf (in bytes)

Returns: number of bytes written on success, negative error code on failure

int **rados_read**(rados_ioctx_t *io*, const char **oid*, char **buf*, size_t *len*, uint64_t *off*)

Read data from an object.

The io context determines the snapshot to read from, if any was set by **rados_ioctx_snap_set_read()**.

Parameters:
- **io** – the context in which to perform the read
- **oid** – the name of the object to read from
- **buf** – where to store the results
- **len** – the number of bytes to read
- **off** – the offset to start reading from in the object

Returns: number of bytes read on success, negative error code on failure

int **rados_remove**(rados_ioctx_t *io*, const char **oid*)

Delete an object.

> **Note:** This does not delete any snapshots of the object.

Parameters:
- **io** – the pool to delete the object from
- **oid** – the name of the object to delete

Returns: 0 on success, negative error code on failure

int **rados_trunc**(rados_ioctx_t *io*, const char **oid*, uint64_t *size*)

Resize an object.

If this enlarges the object, the new area is logically filled with zeroes. If this shrinks the object, the excess data is removed.

Parameters:
- **io** – the context in which to truncate
- **oid** – the name of the object
- **size** – the new size of the object in bytes

Returns: 0 on success, negative error code on failure

int **rados_getxattr**(rados_ioctx_t *io*, const char **o*, const char **name*, char **buf*, size_t *len*)

Get the value of an extended attribute on an object.

Parameters:
- **io** – the context in which the attribute is read
- **o** – name of the object
- **name** – which extended attribute to read
- **buf** – where to store the result
- **len** – size of buf in bytes

Returns: length of xattr value on success, negative error code on failure

int **rados_setxattr**(rados_ioctx_t *io*, const char **o*, const char **name*, const char **buf*, size_t *len*)

Set an extended attribute on an object.

Parameters:
- **io** – the context in which xattr is set
- **o** – name of the object
- **name** – which extended attribute to set
- **buf** – what to store in the xattr
- **len** – the number of bytes in buf

Returns: 0 on success, negative error code on failure

int **rados_rmxattr**(rados_ioctx_t *io*, const char **o*, const char **name*)

Delete an extended attribute from an object.

Parameters:
- **io** – the context in which to delete the xattr
- **o** – the name of the object
- **name** – which xattr to delete

Returns: 0 on success, negative error code on failure

int **rados_getxattrs**(rados_ioctx_t *io*, const char **oid*, rados_xattrs_iter_t **iter*)

Start iterating over xattrs on an object.

Postcondition: iter is a valid iterator

Parameters:
- **io** – the context in which to list xattrs
- **oid** – name of the object
- **iter** – where to store the iterator

Returns: 0 on success, negative error code on failure

int **rados_getxattrs_next**(rados_xattrs_iter_t *iter*, const char ***name*, const char ***val*, size_t **len*)

Get the next xattr on the object.

Precondition: iter is a valid iterator

Postcondition: name is the NULL-terminated name of the next xattr, and val contains the value of the xattr, which is of length len. If the end of the list has been reached, name and val are NULL, and len is 0.

Parameters:
- **iter** – iterator to advance

- **name** – where to store the name of the next xattr
- **val** – where to store the value of the next xattr
- **len** – the number of bytes in val

Returns: 0 on success, negative error code on failure

void **rados_getxattrs_end**(rados_xattrs_iter_t *iter*)

Close the xattr iterator.

iter should not be used after this is called.

Parameters:
- **iter** – the iterator to close

int **rados_stat**(rados_ioctx_t *io*, const char **o*, uint64_t **psize*, time_t **pmtime*)

Get object stats (size/mtime)

TODO: when are these set, and by whom? can they be out of date?

Parameters:
- **io** – ioctx
- **o** – object name
- **psize** – where to store object size
- **pmtime** – where to store modification time

Returns: 0 on success, negative error code on failure

int **rados_tmap_update**(rados_ioctx_t *io*, const char **o*, const char **cmdbuf*, size_t *cmdbuflen*)

Update tmap (trivial map)

Do compound update to a tmap object, inserting or deleting some number of records. cmdbuf is a series of operation byte codes, following by command payload. Each command is a single-byte command code, whose value is one of CEPH_OSD_TMAP_*.

- update tmap 'header'
  - 1 byte = CEPH_OSD_TMAP_HDR
  - 4 bytes = data length (little endian)
  - N bytes = data
- insert/update one key/value pair
  - 1 byte = CEPH_OSD_TMAP_SET
  - 4 bytes = key name length (little endian)
  - N bytes = key name
  - 4 bytes = data length (little endian)
  - M bytes = data
- insert one key/value pair; return -EEXIST if it already exists.
  - 1 byte = CEPH_OSD_TMAP_CREATE
  - 4 bytes = key name length (little endian)
  - N bytes = key name
  - 4 bytes = data length (little endian)
  - M bytes = data
- remove one key/value pair
  - 1 byte = CEPH_OSD_TMAP_RM
  - 4 bytes = key name length (little endian)
  - N bytes = key name

Restrictions:

- The HDR update must preceed any key/value updates.
- All key/value updates must be in lexicographically sorted order in cmdbuf.
- You can read/write to a tmap object via the regular APIs, but you should be careful not to corrupt it. Also be aware that the object format may change without notice.

Parameters:
- **io** – ioctx
- **o** – object name
- **cmdbuf** – command buffer
- **cmdbuflen** – command buffer length in bytes

Returns: 0 on success, negative error code on failure

int **rados_tmap_put**(rados_ioctx_t *io*, const char **o*, const char **buf*, size_t *buflen*)

Store complete tmap (trivial map) object.

Put a full tmap object into the store, replacing what was there.

The format of buf is:

- 4 bytes - length of header (little endian)
- N bytes - header data
- 4 bytes - number of keys (little endian)

and for each key,

- 4 bytes - key name length (little endian)
- N bytes - key name
- 4 bytes - value length (little endian)
- M bytes - value data

Parameters:
- **io** – ioctx
- **o** – object name
- **buf** – buffer
- **buflen** – buffer length in bytes

Returns: 0 on success, negative error code on failure

int **rados_tmap_get**(rados_ioctx_t *io*, const char **o*, char **buf*, size_t *buflen*)

Fetch complete tmap (trivial map) object.

Read a full tmap object. See **rados_tmap_put()** for the format the data is returned in.

Parameters:
- **io** – ioctx
- **o** – object name
- **buf** – buffer
- **buflen** – buffer length in bytes

Returns: 0 on success, negative error code on failure
-ERANGE if buf isn't big enough

int **rados_exec**(rados_ioctx_t *io*, const char **oid*, const char **cls*, const char **method*, const char **in_buf*, size_t *in_len*, char **buf*, size_t *out_len*)

Execute an **OSD** class method on an object.

The **OSD** has a plugin mechanism for performing complicated operations on an object atomically. These plugins are called classes. This function allows librados users to call the custom methods. The input and output formats are defined by the class. Classes in ceph.git can be found in src/cls_*.cc

Parameters:
- **io** – the context in which to call the method
- **oid** – the object to call the method on
- **cls** – the name of the class
- **method** – the name of the method
- **in_buf** – where to find input
- **in_len** – length of in_buf in bytes
- **buf** – where to store output
- **out_len** – length of buf in bytes

Returns: the length of the output, or -ERANGE if out_buf does not have enough space to store it (For methods that return data). For methods that don't return data, the return value is method-specific.

int **rados_aio_create_completion**(void **cb_arg*, rados_callback_t *cb_complete*, rados_callback_t *cb_safe*, rados_completion_t **pc*)

Constructs a completion to use with asynchronous operations.

The complete and safe callbacks correspond to operations being acked and committed, respectively. The callbacks are called in order of receipt, so the safe callback may be triggered before the complete callback, and vice versa. This is affected by journalling on the OSDs.

TODO: more complete documentation of this elsewhere (in the RADOS docs?)

Note: Read operations only get a complete callback.
BUG: this should check for ENOMEM instead of throwing an exception

Parameters:
- **cb_arg** – application-defined data passed to the callback functions
- **cb_complete** – the function to be called when the operation is in memory on all relpicas
- **cb_safe** – the function to be called when the operation is on stable storage on all replicas
- **pc** – where to store the completion

Returns: 0

int **rados_aio_wait_for_complete**(rados_completion_t *c*)

Block until an operation completes.

This means it is in memory on all replicas.

Note: BUG: this should be void

Parameters:
- **c** – operation to wait for

Returns: 0

int **rados_aio_wait_for_safe**(rados_completion_t *c*)

Block until an operation is safe.

This means it is on stable storage on all replicas.

Note: BUG: this should be void

Parameters:
- **c** – operation to wait for

Returns: 0

int **rados_aio_is_complete**(rados_completion_t *c*)

Has an asynchronous operation completed?

Warning: This does not imply that the complete callback has finished

Parameters:
- **c** – async operation to inspect

Returns: whether c is complete

int **rados_aio_is_safe**(rados_completion_t *c*)

Is an asynchronous operation safe?

Warning: This does not imply that the safe callback has finished

Parameters:
- **c** – async operation to inspect

Returns: whether c is safe

int **rados_aio_wait_for_complete_and_cb**(rados_completion_t *c*)

Block until an operation completes and callback completes.

This means it is in memory on all replicas and can be read.

Note: BUG: this should be void

Parameters:
- **c** – operation to wait for

Returns: 0

int **rados_aio_wait_for_safe_and_cb**(rados_completion_t *c*)

Block until an operation is safe and callback has completed.

This means it is on stable storage on all replicas.

Note: BUG: this should be void

Parameters:
- **c** – operation to wait for

Returns: 0

int **rados_aio_is_complete_and_cb**(rados_completion_t *c*)

Has an asynchronous operation and callback completed.

Parameters:
- **c** – async operation to inspect

Returns: whether c is complete

int **rados_aio_is_safe_and_cb**(rados_completion_t *c*)

Is an asynchronous operation safe and has the callback completed.

Parameters:
- **c** – async operation to inspect

Returns: whether c is safe

int **rados_aio_get_return_value**(rados_completion_t *c*)

Get the return value of an asychronous operation.

The return value is set when the operation is complete or safe, whichever comes first.

Precondition: The operation is safe or complete

Note: BUG: complete callback may never be called when the safe message is received before the complete message

Parameters:
- **c** – async operation to inspect

Returns: return value of the operation

void **rados_aio_release**(rados_completion_t *c*)

Release a completion.

Call this when you no longer need the completion. It may not be freed immediately if the operation is not acked and committed.

Parameters:
- **c** – completion to release

int **rados_aio_write**(rados_ioctx_t *io*, const char **oid*, rados_completion_t *completion*, const char **buf*, size_t *len*, uint64_t *off*)

Write data to an object asynchronously.

Queues the write and returns. The return value of the completion will be 0 on success, negative error code on failure.

Parameters:
- **io** – the context in which the write will occur
- **oid** – name of the object
- **completion** – what to do when the write is safe and complete
- **buf** – data to write
- **len** – length of the data, in bytes
- **off** – byte offset in the object to begin writing at

Returns: 0 on success, -EROFS if the io context specifies a snap_seq other than LIBRADOS_SNAP_HEAD

int **rados_aio_append**(rados_ioctx_t *io*, const char **oid*, rados_completion_t *completion*, const char **buf*, size_t *len*)

Asychronously append data to an object.

Queues the append and returns.

The return value of the completion will be 0 on success, negative error code on failure.

Parameters:
- **io** – the context to operate in
- **oid** – the name of the object
- **completion** – what to do when the append is safe and complete
- **buf** – the data to append
- **len** – length of buf (in bytes)

Returns: 0 on success, -EROFS if the io context specifies a snap_seq other than LIBRADOS_SNAP_HEAD

int **rados_aio_write_full**(rados_ioctx_t *io*, const char **oid*, rados_completion_t *completion*, const char **buf*, size_t *len*)

Asychronously write an entire object.

The object is filled with the provided data. If the object exists, it is atomically truncated and then written. Queues the write_full and returns.

The return value of the completion will be 0 on success, negative error code on failure.

Parameters:
- **io** – the io context in which the write will occur
- **oid** – name of the object
- **completion** – what to do when the write_full is safe and complete
- **buf** – data to write
- **len** – length of the data, in bytes

Returns: 0 on success, -EROFS if the io context specifies a snap_seq other than LIBRADOS_SNAP_HEAD

int **rados_aio_remove**(rados_ioctx_t *io*, const char **oid*, rados_completion_t *completion*)

Asychronously remove an object.

Queues the remove and returns.

The return value of the completion will be 0 on success, negative error code on failure.

Parameters:
- **io** – the context to operate in
- **oid** – the name of the object
- **completion** – what to do when the remove is safe and complete

Returns: 0 on success, -EROFS if the io context specifies a snap_seq other than LIBRADOS_SNAP_HEAD

int **rados_aio_read**(rados_ioctx_t *io*, const char **oid*, rados_completion_t *completion*, char **buf*, size_t *len*, uint64_t *off*)

Asychronously read data from an object.

The io context determines the snapshot to read from, if any was set by **rados_ioctx_snap_set_read()**.

The return value of the completion will be number of bytes read on success, negative error code on failure.

Note: only the 'complete' callback of the completion will be called.

Parameters:
- **io** – the context in which to perform the read
- **oid** – the name of the object to read from
- **completion** – what to do when the read is complete
- **buf** – where to store the results
- **len** – the number of bytes to read
- **off** – the offset to start reading from in the object

Returns: 0 on success, negative error code on failure

int **rados_aio_flush**(rados_ioctx_t *io*)

Block until all pending writes in an io context are safe.

This is not equivalent to calling **rados_aio_wait_for_safe()**on all write completions, since this waits for the associated callbacks to complete as well.

Note: BUG: always returns 0, should be void or accept a timeout

Parameters:
- **io** – the context to flush

Returns: 0 on success, negative error code on failure

int **rados_aio_flush_async**(rados_ioctx_t *io*, rados_completion_t *completion*)

Schedule a callback for when all currently pending aio writes are safe.

This is a non-blocking version of **rados_aio_flush()**.

Parameters:
- **io** – the context to flush
- **completion** – what to do when the writes are safe

Returns: 0 on success, negative error code on failure

int **rados_aio_stat**(rados_ioctx_t *io*, const char *o, rados_completion_t *completion*, uint64_t *psize, time_t *pmtime)

Asynchronously get object stats (size/mtime)

Parameters:
- **io** – ioctx
- **o** – object name
- **psize** – where to store object size
- **pmtime** – where to store modification time

Returns: 0 on success, negative error code on failure

int **rados_watch**(rados_ioctx_t *io*, const char *o, uint64_t *ver*, uint64_t *handle, rados_watchcb_t *watchcb*, void *arg)

Register an interest in an object.

A watch operation registers the client as being interested in notifications on an object. OSDs keep track of watches on persistent storage, so they are preserved across cluster changes by the normal recovery process. If the client loses its connection to the primary **OSD**for a watched object, the watch will be removed after 30 seconds. Watches are automatically reestablished when a new connection is made, or a placement group switches OSDs.

Note: BUG: watch timeout should be configurable
BUG: librados should provide a way for watchers to notice connection resets

BUG: the ver parameter does not work, and -ERANGE will never be returned (

Warning: asphyxiate: No renderer found for doxygen tag ‘ulink’

```
<ulink xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" url="http://www.t
```

)

Parameters:
- **io** – the pool the object is in
- **o** – the object to watch
- **ver** – expected version of the object
- **handle** – where to store the internal id assigned to this watch
- **watchcb** – what to do when a notify is received on this object
- **arg** – application defined data to pass when watchcb is called

Returns: 0 on success, negative error code on failure
-ERANGE if the version of the object is greater than ver

int **rados_unwatch**(rados_ioctx_t *io*, const char *o, uint64_t *handle*)

Unregister an interest in an object.

Once this completes, no more notifies will be sent to us for this watch. This should be called to clean up unneeded watchers.

Parameters:
- **io** – the pool the object is in

- **o** – the name of the watched object
- **handle** – which watch to unregister

Returns: 0 on success, negative error code on failure

int **rados_notify**(rados_ioctx_t *io*, const char **o*, uint64_t *ver*, const char **buf*, int *buf_len*)

Sychronously notify watchers of an object.

This blocks until all watchers of the object have received and reacted to the notify, or a timeout is reached.

> **Note:** BUG: the timeout is not changeable via the C API
> BUG: the bufferlist is inaccessible in a rados_watchcb_t

Parameters:
- **io** – the pool the object is in
- **o** – the name of the object
- **ver** – obsolete - just pass zero
- **buf** – data to send to watchers
- **buf_len** – length of buf in bytes

Returns: 0 on success, negative error code on failure