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 will 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_aio_wait_for_complete(comp); // in memory
rados_aio_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

Defines

LIBRADOS_ALL_NSPACES

Pass as nspace argument to rados_ioctx_set_namespace() before calling rados_nobjects_list_open() to return all objects in all namespaces.

struct **obj_watch_t**

#include <rados_types.h>
One item from list_watchers

Public Members

char obj_watch_t::addr[256]

int64_t **watcher_id**

uint64_t **cookie**

uint32_t **timeout_seconds**

xattr comparison operations

Operators for comparing xattrs on objects, and aborting the rados_read_op or rados_write_op transaction if the comparison fails.

enum **[anonymous]**

Values:

LIBRADOS_CMPXATTR_OP_EQ = 1

LIBRADOS_CMPXATTR_OP_NE = 2

LIBRADOS_CMPXATTR_OP_GT = 3

LIBRADOS_CMPXATTR_OP_GTE = 4

LIBRADOS_CMPXATTR_OP_LT = 5

LIBRADOS_CMPXATTR_OP_LTE = 6

Operation Flags

Flags for rados_read_op_operate(), rados_write_op_operate(), rados_aio_read_op_operate(), and rados_aio_write_op_operate(). See librados.hpp for details.

enum **[anonymous]**

Values:

LIBRADOS_OPERATION_NOFLAG = 0

LIBRADOS_OPERATION_BALANCE_READS = 1

LIBRADOS_OPERATION_LOCALIZE_READS = 2

LIBRADOS_OPERATION_ORDER_READS_WRITES = 4

LIBRADOS_OPERATION_IGNORE_CACHE = 8

LIBRADOS_OPERATION_SKIPRWLOCKS = 16

LIBRADOS_OPERATION_IGNORE_OVERLAY = 32

LIBRADOS_OPERATION_FULL_TRY = 64

LIBRADOS_OPERATION_FULL_FORCE = 128

LIBRADOS_OPERATION_IGNORE_REDIRECT = 256

Alloc hint flags

Flags for rados_write_op_alloc_hint2() and rados_set_alloc_hint2() indicating future IO patterns.

enum **[anonymous]**

Values:

LIBRADOS_ALLOC_HINT_FLAG_SEQUENTIAL_WRITE = 1

LIBRADOS_ALLOC_HINT_FLAG_RANDOM_WRITE = 2

LIBRADOS_ALLOC_HINT_FLAG_SEQUENTIAL_READ = 4

LIBRADOS_ALLOC_HINT_FLAG_RANDOM_READ = 8

LIBRADOS_ALLOC_HINT_FLAG_APPEND_ONLY = 16

LIBRADOS_ALLOC_HINT_FLAG_IMMUTABLE = 32

LIBRADOS_ALLOC_HINT_FLAG_SHORTLIVED = 64

LIBRADOS_ALLOC_HINT_FLAG_LONGLIVED = 128

LIBRADOS_ALLOC_HINT_FLAG_COMPRESSIBLE = 256

LIBRADOS_ALLOC_HINT_FLAG_INCOMPRESSIBLE = 512

Asynchronous I/O

Read and write to objects without blocking.

typedef **`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

CEPH_RADOS_API 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.

Note

BUG: this should check for ENOMEM instead of throwing an exception

Return

0

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

CEPH_RADOS_API 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

Return

0

Parameters

- `c`: operation to wait for

CEPH_RADOS_API 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

Return

0

Parameters

- `c`: operation to wait for

CEPH_RADOS_API 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

Return

whether c is complete

Parameters

- c: async operation to inspect

CEPH_RADOS_API 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

Return

whether c is safe

Parameters

- c: async operation to inspect

CEPH_RADOS_API 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

Return

0

Parameters

- c: operation to wait for

CEPH_RADOS_API 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

Return

0

Parameters

- c: operation to wait for

CEPH_RADOS_API int **`rados_aio_is_complete_and_cb`**(rados_completion_t *c*)

Has an asynchronous operation and callback completed

Return

whether c is complete

Parameters

- c: async operation to inspect

CEPH_RADOS_API int **`rados_aio_is_safe_and_cb`**(rados_completion_t *c*)

Is an asynchronous operation safe and has the callback completed

Return

whether c is safe

Parameters

- c: async operation to inspect

CEPH_RADOS_API 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.

Pre

The operation is safe or complete

Note

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

Return

return value of the operation

Parameters

- c: async operation to inspect

CEPH_RADOS_API uint64_t **rados_aio_get_version**(rados_completion_t *c*)

Get the internal object version of the target of an asychronous operation

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

Pre

The operation is safe or complete

Note

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

Return

version number of the asychronous operation's target

Parameters

- c: async operation to inspect

CEPH_RADOS_API 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

CEPH_RADOS_API 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.

Return

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

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

CEPH_RADOS_API 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.

Return

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

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)

CEPH_RADOS_API 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.

Return

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

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

CEPH_RADOS_API int **`rados_aio_writesame`**(rados_ioctx_t *io*, const char * *oid*, rados_completion_t *completion*, const char * *buf*, size_t *data_len*, size_t *write_len*, uint64_t *off*)

Asychronously write the same buffer multiple times

Queues the writesame and returns.

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

Return

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

Parameters

- `io`: the io context in which the write will occur
- `oid`: name of the object
- `completion`: what to do when the writesame is safe and complete
- `buf`: data to write
- `data_len`: length of the data, in bytes
- `write_len`: the total number of bytes to write
- `off`: byte offset in the object to begin writing at

CEPH_RADOS_API 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.

Return

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

Parameters

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

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API 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

Return

0 on success, negative error code on failure

Parameters

- `io`: the context to flush

CEPH_RADOS_API 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().

Return

0 on success, negative error code on failure

Parameters

- `io`: the context to flush
- `completion`: what to do when the writes are safe

CEPH_RADOS_API 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)

Return

0 on success, negative error code on failure

Parameters

- `io`: ioctx
- `o`: object name
- `psize`: where to store object size
- `pmtime`: where to store modification time

CEPH_RADOS_API int **`rados_aio_cmpext`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *cmp_buf*, size_t *cmp_len*, uint64_t *off*)

Asynchronously compare an on-disk object range with a buffer

Return

0 on success, negative error code on failure, (-MAX_ERRNO - mismatch_off) on mismatch

Parameters

- `io`: the context in which to perform the comparison
- `o`: the name of the object to compare with
- `completion`: what to do when the comparison is complete
- `cmp_buf`: buffer containing bytes to be compared with object contents
- `cmp_len`: length to compare and size of in bytes
- `off`: object byte offset at which to start the comparison

CEPH_RADOS_API int **`rados_aio_cancel`**(rados_ioctx_t *io*, rados_completion_t *completion*)

Cancel async operation

Return

0 on success, negative error code on failure

Parameters

- `io`: ioctx
- `completion`: completion handle

CEPH_RADOS_API int **`rados_aio_exec`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *cls*, const char * *method*, const char * *in_buf*, size_t *in_len*, char * *buf*, size_t *out_len*)

Asynchronously 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 subdirectories

Return

0 on success, negative error code on failure

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

Watch/Notify

Watch/notify is a protocol to help communicate among clients. It can be used to sychronize client state. All that's needed is a well-known object name (for example, rbd uses the header object of an image).

Watchers register an interest in an object, and receive all notifies on that object. A notify attempts to communicate with all clients watching an object, and blocks on the notifier until each client responds or a timeout is reached.

See rados_watch() and rados_notify() for more details.

typedef **`rados_watchcb_t`**

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

Note

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

Note

BUG: ver is unused

Parameters

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

typedef **`rados_watchcb2_t`**

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

Parameters

- `arg`: opaque user-defined value provided to rados_watch2()
- `notify_id`: an id for this notify event
- `handle`: the watcher handle we are notifying
- `notifier_id`: the unique client id for the notifier
- `data`: payload from the notifier
- `datalen`: length of payload buffer

typedef **`rados_watcherrcb_t`**

Callback activated when we encounter an error with the watch session. This can happen when the location of the objects moves within the cluster and we fail to register our watch with the new object location, or when our connection with the object OSD is otherwise interrupted and we may have missed notify events.

Parameters

- `pre`: opaque user-defined value provided to rados_watch2()

- `err`: error code

CEPH_RADOS_API int **`rados_watch`**(rados_ioctx_t *io*, const char * *o*, uint64_t *ver*, uint64_t * *cookie*, 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: librados should provide a way for watchers to notice connection resets

Note

BUG: the ver parameter does not work, and -ERANGE will never be returned (See URL tracker.ceph.com/issues/2592)

Return

0 on success, negative error code on failure

Return

-ERANGE if the version of the object is greater than ver

Parameters

- `io`: the pool the object is in
- `o`: the object to watch
- `ver`: expected version of the object
- `cookie`: 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

CEPH_RADOS_API int **`rados_watch2`**(rados_ioctx_t *io*, const char * *o*, uint64_t * *cookie*, rados_watchcb2_t *watchcb*, rados_watcherrcb_t *watcherrcb*, 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 a timeout configured with osd_client_watch_timeout. Watches are automatically reestablished when a new connection is made, or a placement group switches OSDs.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the object to watch
- `cookie`: where to store the internal id assigned to this watch
- `watchcb`: what to do when a notify is received on this object
- `watcherrcb`: what to do when the watch session encounters an error
- `arg`: opaque value to pass to the callback

CEPH_RADOS_API int **`rados_watch3`**(rados_ioctx_t *io*, const char * *o*, uint64_t * *cookie*, rados_watchcb2_t *watchcb*, rados_watcherrcb_t *watcherrcb*, uint32_t *timeout*, 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. Watches are automatically reestablished when a new connection is made, or a placement group switches OSDs.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in

- `o`: the object to watch
- `cookie`: where to store the internal id assigned to this watch
- `watchcb`: what to do when a notify is received on this object
- `watcherrcb`: what to do when the watch session encounters an error
- `timeout`: how many seconds the connection will keep after disconnection
- `arg`: opaque value to pass to the callback

CEPH_RADOS_API int **`rados_aio_watch`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, uint64_t * *handle*, rados_watchcb2_t *watchcb*, rados_watcherrcb_t *watcherrcb*, void * *arg*)

Asynchronous 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.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the object to watch
- `completion`: what to do when operation has been attempted
- `handle`: where to store the internal id assigned to this watch
- `watchcb`: what to do when a notify is received on this object
- `watcherrcb`: what to do when the watch session encounters an error
- `arg`: opaque value to pass to the callback

CEPH_RADOS_API int **`rados_aio_watch2`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, uint64_t * *handle*, rados_watchcb2_t *watchcb*, rados_watcherrcb_t *watcherrcb*, uint32_t *timeout*, void * *arg*)

Asynchronous 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 the number of seconds that configured in timeout parameter. Watches are automatically reestablished when a new connection is made, or a placement group switches OSDs.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the object to watch
- `completion`: what to do when operation has been attempted
- `handle`: where to store the internal id assigned to this watch
- `watchcb`: what to do when a notify is received on this object
- `watcherrcb`: what to do when the watch session encounters an error
- `timeout`: how many seconds the connection will keep after disconnection
- `arg`: opaque value to pass to the callback

CEPH_RADOS_API int **`rados_watch_check`**(rados_ioctx_t *io*, uint64_t *cookie*)

Check on the status of a watch

Return the number of milliseconds since the watch was last confirmed. Or, if there has been an error, return that.

If there is an error, the watch is no longer valid, and should be destroyed with rados_unwatch2(). The the user is still interested in the object, a new watch should be created with rados_watch2().

Return

ms since last confirmed on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `cookie`: the watch handle

CEPH_RADOS_API int **`rados_unwatch`**(rados_ioctx_t *io*, const char * *o*, uint64_t *cookie*)

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.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the name of the watched object (ignored)
- `cookie`: which watch to unregister

CEPH_RADOS_API int **`rados_unwatch2`**(rados_ioctx_t *io*, uint64_t *cookie*)

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.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `cookie`: which watch to unregister

CEPH_RADOS_API int **`rados_aio_unwatch`**(rados_ioctx_t *io*, uint64_t *cookie*, rados_completion_t *completion*)

Asynchronous 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.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `completion`: what to do when operation has been attempted
- `cookie`: which watch to unregister

CEPH_RADOS_API 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

Note

BUG: the bufferlist is inaccessible in a rados_watchcb_t

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API int **`rados_notify2`**(rados_ioctx_t *io*, const char * *o*, const char * *buf*, int *buf_len*, uint64_t *timeout_ms*, char ** *reply_buffer*, size_t * *reply_buffer_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.

The reply buffer is optional. If specified, the client will get back an encoded buffer that includes the ids of the clients that acknowledged the notify as well as their notify ack payloads (if any). Clients that timed out are not included. Even clients that do not include a notify ack payload are included in the list but have a 0-length payload associated with them. The format:

le32 num_acks { le64 gid global id for the client (for client.1234 that's 1234) le64 cookie cookie for the client le32 buflen length of reply message buffer u8 * buflen payload } * num_acks le32 num_timeouts { le64 gid global id for the client le64 cookie cookie for the client } * num_timeouts

Note: There may be multiple instances of the same gid if there are multiple watchers registered via the same client.

Note: The buffer must be released with rados_buffer_free() when the user is done with it.

Note: Since the result buffer includes clients that time out, it will be set even when rados_notify() returns an error code (like -ETIMEDOUT).

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `completion`: what to do when operation has been attempted
- `o`: the name of the object
- `buf`: data to send to watchers
- `buf_len`: length of buf in bytes
- `timeout_ms`: notify timeout (in ms)
- `reply_buffer`: pointer to reply buffer pointer (free with rados_buffer_free)
- `reply_buffer_len`: pointer to size of reply buffer

CEPH_RADOS_API int **`rados_aio_notify`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *buf*, int *buf_len*, uint64_t *timeout_ms*, char ** *reply_buffer*, size_t * *reply_buffer_len*)

CEPH_RADOS_API int **`rados_notify_ack`**(rados_ioctx_t *io*, const char * *o*, uint64_t *notify_id*, uint64_t *cookie*, const char * *buf*, int *buf_len*)

Acknolwedge receipt of a notify

Return

0 on success

Parameters

- `io`: the pool the object is in
- `o`: the name of the object
- `notify_id`: the notify_id we got on the watchcb2_t callback
- `cookie`: the watcher handle
- `buf`: payload to return to notifier (optional)
- `buf_len`: payload length

CEPH_RADOS_API int **`rados_watch_flush`**(rados_t *cluster*)

Flush watch/notify callbacks

This call will block until all pending watch/notify callbacks have been executed and the queue is empty. It should usually be called after shutting down any watches before shutting down the ioctx or librados to ensure that any callbacks do not misuse the ioctx (for example by calling rados_notify_ack after the ioctx has been destroyed).

Parameters

- `cluster`: the cluster handle

CEPH_RADOS_API int **`rados_aio_watch_flush`**(rados_t *cluster*, rados_completion_t *completion*)

Flush watch/notify callbacks

This call will be nonblock, and the completion will be called until all pending watch/notify callbacks have been executed and the queue is empty. It should usually be called after shutting down any watches before shutting down the ioctx or librados to ensure that any callbacks do not misuse the ioctx (for example by calling rados_notify_ack after the ioctx has been destroyed).

Parameters

- `cluster`: the cluster handle
- `completion`: what to do when operation has been attempted

Mon/OSD/PG Commands

These interfaces send commands relating to the monitor, OSD, or PGs.

typedef void **(* rados_log_callback_t)**(void **arg*, const char **line*, const char **who*, uint64_t *sec*, uint64_t *nsec*, uint64_t *seq*, const char **level*, const char **msg*)

typedef void **(* rados_log_callback2_t)**(void **arg*, const char **line*, const char **channel*, const char **who*, const char **name*, uint64_t *sec*, uint64_t *nsec*, uint64_t *seq*, const char **level*, const char **msg*)

CEPH_RADOS_API int **rados_mon_command**(rados_t *cluster*, const char ** *cmd*, size_t *cmdlen*, const char * *inbuf*, size_t *inbuflen*, char ** *outbuf*, size_t * *outbuflen*, char ** *outs*, size_t * *outslen*)

Send monitor command.

The result buffers are allocated on the heap; the caller is expected to release that memory with rados_buffer_free(). The buffer and length pointers can all be NULL, in which case they are not filled in.

Note

Takes command string in carefully-formatted JSON; must match defined commands, types, etc.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle
- `cmd`: an array of char *'s representing the command
- `cmdlen`: count of valid entries in cmd
- `inbuf`: any bulk input data (crush map, etc.)
- `outbuf`: double pointer to output buffer
- `outbuflen`: pointer to output buffer length
- `outs`: double pointer to status string
- `outslen`: pointer to status string length

CEPH_RADOS_API int **rados_mgr_command**(rados_t *cluster*, const char ** *cmd*, size_t *cmdlen*, const char * *inbuf*, size_t *inbuflen*, char ** *outbuf*, size_t * *outbuflen*, char ** *outs*, size_t * *outslen*)

Send ceph-mgr command.

The result buffers are allocated on the heap; the caller is expected to release that memory with rados_buffer_free(). The buffer and length pointers can all be NULL, in which case they are not filled in.

Note

Takes command string in carefully-formatted JSON; must match defined commands, types, etc.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle
- `cmd`: an array of char *'s representing the command
- `cmdlen`: count of valid entries in cmd
- `inbuf`: any bulk input data (crush map, etc.)
- `outbuf`: double pointer to output buffer
- `outbuflen`: pointer to output buffer length
- `outs`: double pointer to status string
- `outslen`: pointer to status string length

CEPH_RADOS_API int **rados_mon_command_target**(rados_t *cluster*, const char * *name*, const char ** *cmd*, size_t *cmdlen*, const char * *inbuf*, size_t *inbuflen*, char ** *outbuf*, size_t * *outbuflen*, char ** *outs*, size_t * *outslen*)

Send monitor command to a specific monitor.

The result buffers are allocated on the heap; the caller is expected to release that memory with rados_buffer_free(). The buffer and length pointers can all be NULL, in which case they are not filled in.

Note

Takes command string in carefully-formatted JSON; must match defined commands, types, etc.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle
- `name`: target monitor's name
- `cmd`: an array of char *'s representing the command
- `cmdlen`: count of valid entries in cmd
- `inbuf`: any bulk input data (crush map, etc.)
- `outbuf`: double pointer to output buffer
- `outbuflen`: pointer to output buffer length
- `outs`: double pointer to status string
- `outslen`: pointer to status string length

CEPH_RADOS_API void **`rados_buffer_free`**(char * *buf*)

free a rados-allocated buffer

Release memory allocated by librados calls like rados_mon_command().

Parameters

- `buf`: buffer pointer

CEPH_RADOS_API int **`rados_osd_command`**(rados_t *cluster*, int *osdid*, const char ** *cmd*, size_t *cmdlen*, const char * *inbuf*, size_t *inbuflen*, char ** *outbuf*, size_t * *outbuflen*, char ** *outs*, size_t * *outslen*)

CEPH_RADOS_API int **`rados_pg_command`**(rados_t *cluster*, const char * *pgstr*, const char ** *cmd*, size_t *cmdlen*, const char * *inbuf*, size_t *inbuflen*, char ** *outbuf*, size_t * *outbuflen*, char ** *outs*, size_t * *outslen*)

CEPH_RADOS_API int **`rados_monitor_log`**(rados_t *cluster*, const char * *level*, rados_log_callback_t *cb*, void * *arg*)

CEPH_RADOS_API int **`rados_monitor_log2`**(rados_t *cluster*, const char * *level*, rados_log_callback2_t *cb*, void * *arg*)

CEPH_RADOS_API int **`rados_service_register`**(rados_t *cluster*, const char * *service*, const char * *daemon*, const char * *metadata_dict*)

register daemon instance for a service

Register us as a daemon providing a particular service. We identify the service (e.g., 'rgw') and our instance name (e.g., 'rgw.$hostname'). The metadata is a map of keys and values with arbitrary static metdata for this instance. The encoding is a series of NULL-terminated strings, alternating key names and values, terminating with an empty key name. For example, "foo\0bar\0this\0that\0\0" is the dict {foo=bar,this=that}.

For the lifetime of the librados instance, regular beacons will be sent to the cluster to maintain our registration in the service map.

Parameters

- `cluster`: handle
- `service`: service name
- `daemon`: daemon instance name
- `metadata_dict`: static daemon metadata dict

CEPH_RADOS_API int **`rados_service_update_status`**(rados_t *cluster*, const char * *status_dict*)

update daemon status

Update our mutable status information in the service map.

The status dict is encoded the same way the daemon metadata is encoded for rados_service_register. For example, "foo\0bar\0this\0that\0\0" is {foo=bar,this=that}.

Parameters

- `cluster`: rados cluster handle
- `status_dict`: status dict

Setup and Teardown

These are the first and last functions to that should be called when using librados.

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API int **rados_create2**(rados_t * *pcluster*, const char *const *clustername*, const char *const *name*, uint64_t *flags*)

Extended version of rados_create.

Like rados_create, but 1) don’t assume ‘client.’+id; allow full specification of name 2) allow specification of cluster name 3) flags for future expansion

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: where to store the handle
- `cct`: the existing configuration to use

CEPH_RADOS_API int **rados_ping_monitor**(rados_t *cluster*, const char * *mon_id*, char ** *outstr*, size_t * *outstrlen*)

Ping the monitor with ID mon_id, storing the resulting reply in buf (if specified) with a maximum size of len.

The result buffer is allocated on the heap; the caller is expected to release that memory with rados_buffer_free(). The buffer and length pointers can be NULL, in which case they are not filled in.

Parameters

- `cluster`: cluster handle
- `mon_id`: ID of the monitor to ping
- `outstr`: double pointer with the resulting reply
- `outstrlen`: pointer with the size of the reply in outstr

CEPH_RADOS_API 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.

Pre

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

Post

If this succeeds, any function in librados may be used

Return

0 on sucess, negative error code on failure

Parameters

- `cluster`: The cluster to connect to.

CEPH_RADOS_API 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 rados_aio_flush() on all open io contexts.

Warning

We implicitly call rados_watch_flush() on shutdown. If there are watches being used, this should be done explicitly before destroying the relevant IoCtx. We do it here as a safety measure.

Post

the cluster handle cannot be used again

Parameters

- `cluster`: the cluster to shutdown

Configuration

These functions read and update Ceph configuration for a cluster handle. Any configuration changes must be done before connecting to the cluster.

Options that librados users might want to set include:

- mon_host
- auth_supported
- key, keyfile, or keyring when using cephx
- log_file, log_to_stderr, err_to_stderr, and log_to_syslog
- debug_rados, debug_objecter, debug_monc, debug_auth, or debug_ms

See docs.ceph.com for information about available configuration options`

CEPH_RADOS_API 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)

Pre

rados_connect() has not been called on the cluster handle

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle to configure
- `path`: path to a Ceph configuration file

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

- mon-host 10.0.0.1:6789
- mon_host 10.0.0.1:6789
- -m 10.0.0.1:6789

Pre

rados_connect() has not been called on the cluster handle

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle to configure
- `argc`: number of arguments in argv
- `argv`: arguments to parse

CEPH_RADOS_API int **`rados_conf_parse_argv_remainder`**(rados_t *cluster*, int *argc*, const char ** *argv*, const char ** *remargv*)

Configure the cluster handle with command line arguments, returning any remainders. Same rados_conf_parse_argv, except for extra remargv argument to hold returns unrecognized arguments.

Pre

rados_connect() has not been called on the cluster handle

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle to configure
- `argc`: number of arguments in argv
- `argv`: arguments to parse
- `remargv`: char* array for returned unrecognized arguments

CEPH_RADOS_API 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.

Pre

rados_connect() has not been called on the cluster handle

Note

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

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle to configure
- `var`: name of the environment variable to read

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

Set a configuration option

Pre

rados_connect() has not been called on the cluster handle

Return

0 on success, negative error code on failure

Return

-ENOENT when the option is not a Ceph configuration option

Parameters

- `cluster`: cluster handle to configure
- `option`: option to set
- `value`: value of the option

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

Get the value of a configuration option

Return

0 on success, negative error code on failure

Return

-ENAMETOOLONG if the buffer is too short to contain the requested value

Parameters

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

Pools

RADOS pools are separate namespaces for objects. Pools may have different crush rules associated with them, so they could have differing replication levels or placement strategies. RADOS permissions are also tied to pools - users can have different read, write, and execute permissions on a per-pool basis.

CEPH_RADOS_API 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.

Buf may be null to determine the buffer size needed to list all pools.

Return

length of the buffer we would need to list all pools

Parameters

- `cluster`: cluster handle
- `buf`: output buffer
- `len`: output buffer length

CEPH_RADOS_API int **`rados_inconsistent_pg_list`**(rados_t *cluster*, int64_t *pool*, char * *buf*, size_t *len*)

List inconsistent placement groups of the given pool

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

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

Return

length of the buffer we would need to list all pools

Parameters

- `cluster`: cluster handle
- `pool`: pool ID
- `buf`: output buffer
- `len`: output buffer length

CEPH_RADOS_API 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.

Return

config handle for this cluster

Parameters

- `cluster`: cluster handle

CEPH_RADOS_API 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

Return

instance global id

Parameters

- `cluster`: cluster handle

CEPH_RADOS_API int **`rados_get_min_compatible_client`**(rados_t *cluster*, int8_t * *min_compat_client*, int8_t * *require_min_compat_client*)

Gets the minimum compatible client version

Return

0 on sucess, negative error code on failure

Parameters

- `cluster`: cluster handle

- `min_compat_client`: minimum compatible client version based upon the current features
- `require_min_compat_client`: required minimum client version based upon explicit setting

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API int **`rados_ioctx_create2`**(rados_t *cluster*, int64_t *pool_id*, rados_ioctx_t * *ioctx*)

CEPH_RADOS_API 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 rados_aio_flush() on the io context before destroying it to do that.

Warning

If this ioctx is used by rados_watch, the caller needs to be sure that all registered watches are disconnected via rados_unwatch() and that rados_watch_flush() is called. This ensures that a racing watch callback does not make use of a destroyed ioctx.

Parameters

- `io`: the io context to dispose of

CEPH_RADOS_API rados_config_t **`rados_ioctx_cct`**(rados_ioctx_t *io*)

Get configuration handle for a pool handle

Return

rados_config_t for this cluster

Parameters

- `io`: pool handle

CEPH_RADOS_API 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_shutdown().

Return

the cluster handle for this io context

Parameters

- `io`: the io context

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `io`: determines which pool to query
- `stats`: where to store the results

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

Get the id of a pool

Return

id of the pool

Return

-ENOENT if the pool is not found

Parameters

- `cluster`: which cluster the pool is in
- `pool_name`: which pool to look up

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

Get the name of a pool

Return

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

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

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: the cluster in which the pool will be created
- `pool_name`: the name of the new pool

CEPH_RADOS_API 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

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API int **`rados_pool_create_with_crush_rule`**(rados_t *cluster*, const char * *pool_name*, uint8_t *crush_rule_num*)

Create a pool with a specific CRUSH rule

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API int **`rados_pool_create_with_all`**(rados_t *cluster*, const char * *pool_name*, uint64_t *auid*, uint8_t *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().

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API int **`rados_pool_get_base_tier`**(rados_t *cluster*, int64_t *pool*, int64_t * *base_tier*)

Returns the pool that is the base tier for this pool.

The return value is the ID of the pool that should be used to read from/write to. If tiering is not set up for the pool, returns `pool`.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: the cluster the pool is in
- `pool`: ID of the pool to query
- `base_tier`: base tier, or `pool` if tiering is not configured

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: the cluster the pool is in
- `pool_name`: which pool to delete

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `io`: reference to the pool to change.
- `auid`: the auid you wish the io to have.

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

Get the auid of a pool

Return

0 on success, negative error code on failure

Parameters

- `io`: pool to query
- `auid`: where to store the auid

CEPH_RADOS_API int **`rados_ioctx_pool_requires_alignment`**(rados_ioctx_t *io*)

CEPH_RADOS_API int **`rados_ioctx_pool_requires_alignment2`**(rados_ioctx_t *io*, int * *requires*)

Test whether the specified pool requires alignment or not.

Return

0 on success, negative error code on failure

Parameters

- `io`: pool to query
- `requires`: 1 if alignment is supported, 0 if not.

CEPH_RADOS_API uint64_t **rados_ioctx_pool_required_alignment**(rados_ioctx_t *io*)

CEPH_RADOS_API int **rados_ioctx_pool_required_alignment2**(rados_ioctx_t *io*, uint64_t * *alignment*)

Get the alignment flavor of a pool

Return

0 on success, negative error code on failure

Parameters

- `io`: pool to query
- `alignment`: where to store the alignment flavor

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

Get the pool id of the io context

Return

the id of the pool the io context uses

Parameters

- `io`: the io context to query

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

Get the pool name of the io context

Return

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

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

Object Locators

CEPH_RADOS_API 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.

Parameters

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

CEPH_RADOS_API void **rados_ioctx_set_namespace**(rados_ioctx_t *io*, const char * *nspace*)

Set the namespace for objects within an io context

The namespace specification further refines a pool into different domains. The mapping of objects to pgs is also based on this value.

Parameters

- `io`: the io context to change
- `nspace`: the name to use as the namespace, or NULL use the default namespace

Listing Objects

CEPH_RADOS_API int **rados_nobjects_list_open**(rados_ioctx_t *io*, rados_list_ctx_t * *ctx*)

Start listing objects in a pool

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool to list from
- `ctx`: the handle to store list context in

CEPH_RADOS_API uint32_t **rados_nobjects_list_get_pg_hash_position**(rados_list_ctx_t *ctx*)

Return hash position of iterator, rounded to the current PG

Return

current hash position, rounded to the current pg

Parameters

- `ctx`: iterator marking where you are in the listing

CEPH_RADOS_API uint32_t **rados_nobjects_list_seek**(rados_list_ctx_t *ctx*, uint32_t *pos*)

Reposition object iterator to a different hash position

Return

actual (rounded) position we moved to

Parameters

- `ctx`: iterator marking where you are in the listing
- `pos`: hash position to move to

CEPH_RADOS_API uint32_t **rados_nobjects_list_seek_cursor**(rados_list_ctx_t *ctx*, rados_object_list_cursor *cursor*)

Reposition object iterator to a different position

Return

rounded position we moved to

Parameters

- `ctx`: iterator marking where you are in the listing
- `cursor`: position to move to

CEPH_RADOS_API int **rados_nobjects_list_get_cursor**(rados_list_ctx_t *ctx*, rados_object_list_cursor * *cursor*)

Reposition object iterator to a different position

The returned handle must be released with rados_object_list_cursor_free().

Return

0 on success, negative error code on failure

Parameters

- `ctx`: iterator marking where you are in the listing
- `cursor`: where to store cursor

CEPH_RADOS_API int **rados_nobjects_list_next**(rados_list_ctx_t *ctx*, const char ** *entry*, const char ** *key*, const char ** *nspace*)

Get the next object name and locator in the pool

*entry and *key are valid until next call to rados_nobjects_list_*

Return

0 on success, negative error code on failure

Return

-ENOENT when there are no more objects to list

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)
- `nspace`: where to store the object namespace (set to NULL to ignore)

CEPH_RADOS_API void **rados_nobjects_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

CEPH_RADOS_API rados_object_list_cursor **`rados_object_list_begin`**(rados_ioctx_t *io*)

Get cursor handle pointing to the *beginning* of a pool.

This is an opaque handle pointing to the start of a pool. It must be released with rados_object_list_cursor_free().

Return

handle for the pool, NULL on error (pool does not exist)

Parameters

- `io`: ioctx for the pool

CEPH_RADOS_API rados_object_list_cursor **`rados_object_list_end`**(rados_ioctx_t *io*)

Get cursor handle pointing to the *end* of a pool.

This is an opaque handle pointing to the start of a pool. It must be released with rados_object_list_cursor_free().

Return

handle for the pool, NULL on error (pool does not exist)

Parameters

- `io`: ioctx for the pool

CEPH_RADOS_API int **`rados_object_list_is_end`**(rados_ioctx_t *io*, rados_object_list_cursor *cur*)

Check if a cursor has reached the end of a pool

Return

1 if the cursor has reached the end of the pool, 0 otherwise

Parameters

- `io`: ioctx
- `cur`: cursor

CEPH_RADOS_API void **`rados_object_list_cursor_free`**(rados_ioctx_t *io*, rados_object_list_cursor *cur*)

Release a cursor

Release a cursor. The handle may not be used after this point.

Parameters

- `io`: ioctx
- `cur`: cursor

CEPH_RADOS_API int **`rados_object_list_cursor_cmp`**(rados_ioctx_t *io*, rados_object_list_cursor *lhs*, rados_object_list_cursor *rhs*)

Compare two cursor positions

Compare two cursors, and indicate whether the first cursor precedes, matches, or follows the second.

Return

-1, 0, or 1 for lhs < rhs, lhs == rhs, or lhs > rhs

Parameters

- `io`: ioctx
- `lhs`: first cursor
- `rhs`: second cursor

CEPH_RADOS_API int **`rados_object_list`**(rados_ioctx_t *io*, const rados_object_list_cursor *start*, const rados_object_list_cursor *finish*, const size_t *result_size*, const char * *filter_buf*, const size_t *filter_buf_len*, rados_object_list_item * *results*, rados_object_list_cursor * *next*)

Return

the number of items set in the result array

CEPH_RADOS_API void **`rados_object_list_free`**(const size_t *result_size*, rados_object_list_item * *results*)

CEPH_RADOS_API void **`rados_object_list_slice`**(rados_ioctx_t *io*, const rados_object_list_cursor *start*, const rados_object_list_cursor *finish*, const size_t *n*, const size_t *m*, rados_object_list_cursor * *split_start*, rados_object_list_cursor * *split_finish*)

Obtain cursors delineating a subset of a range. Use this when you want to split up the work of iterating over the global namespace. Expected use case is when you are iterating in parallel, with `m` workers, and each worker taking an id n.

Parameters

- `start`: start of the range to be sliced up (inclusive)
- `finish`: end of the range to be sliced up (exclusive)
- `m`: how many chunks to divide start-finish into
- `n`: which of the m chunks you would like to get cursors for
- `split_start`: cursor populated with start of the subrange (inclusive)
- `split_finish`: cursor populated with end of the subrange (exclusive)

Snapshots

RADOS snapshots are based upon sequence numbers that form a snapshot context. They are pool-specific. The snapshot context consists of the current snapshot sequence number for a pool, and an array of sequence numbers at which snapshots were taken, in descending order. Whenever a snapshot is created or deleted, the snapshot sequence number for the pool is increased. To add a new snapshot, the new snapshot sequence number must be increased and added to the snapshot context.

There are two ways to manage these snapshot contexts:

1. within the RADOS cluster These are called pool snapshots, and store the snapshot context in the OSDMap. These represent a snapshot of all the objects in a pool.
2. within the RADOS clients These are called self-managed snapshots, and push the responsibility for keeping track of the snapshot context to the clients. For every write, the client must send the snapshot context. In librados, this is accomplished with rados_selfmanaged_snap_set_write_ctx(). These are more difficult to manage, but are restricted to specific objects instead of applying to an entire pool.

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

Create a pool-wide snapshot

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool to snapshot
- `snapname`: the name of the snapshot

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

Delete a pool snapshot

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool to delete the snapshot from
- `snapname`: which snapshot to delete

CEPH_RADOS_API int **`rados_ioctx_snap_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.

Return

0 on success, negative error code on failure

Parameters

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

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

Warning

Deprecated: Use rados_ioctx_snap_rollback() instead

CEPH_RADOS_API 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)

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API void **`rados_aio_ioctx_selfmanaged_snap_create`**(rados_ioctx_t *io*, rados_snap_t * *snapid*, rados_completion_t *completion*)

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API void **`rados_aio_ioctx_selfmanaged_snap_remove`**(rados_ioctx_t *io*, rados_snap_t *snapid*, rados_completion_t *completion*)

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Return

-EINVAL if snaps are not in descending order

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

CEPH_RADOS_API 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.

Return

number of snapshots on success, negative error code on failure

Return

-ERANGE is returned if the snaps array is too short

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

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

Get the id of a pool snapshot

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool to read from
- `name`: the snapshot to find
- `id`: where to store the result

CEPH_RADOS_API 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

Return

0 on success, negative error code on failure

Return

-ERANGE if the name array is too small

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

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

Find when a pool snapshot occurred

Return

0 on success, negative error code on failure

Parameters

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

Synchronous I/O

Writes are replicated to a number of OSDs based on the configuration of the pool they are in. These write functions block until data is in memory on all replicas of the object they're writing to - they are equivalent to doing the corresponding asynchronous write, and the calling rados_ioctx_wait_for_complete(). For greater data safety, use the asynchronous functions and rados_aio_wait_for_safe().

CEPH_RADOS_API 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

Return

last read or written object version

Parameters

- `io`: the io context to check

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

Write *len* bytes from *buf* into the *oid* object, starting at offset *off*. The value of *len* must be <= UINT_MAX/2.

Note

This will never return a positive value not equal to len.

Return

0 on success, negative error code on failure

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

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

Write *len* bytes from *buf* into the *oid* object. The value of *len* must be <= UINT_MAX/2.

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

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API int **`rados_writesame`**(rados_ioctx_t *io*, const char * *oid*, const char * *buf*, size_t *data_len*, size_t *write_len*, uint64_t *off*)

Write the same *data_len* bytes from *buf* multiple times into the *oid* object. *write_len* bytes are written in total, which must be a multiple of *data_len*. The value of *write_len* and *data_len* must be <= UINT_MAX/2.

Return

0 on success, negative error code on failure

Parameters

- `io`: the io context in which the write will occur
- `oid`: name of the object
- `buf`: data to write
- `data_len`: length of the data, in bytes
- `write_len`: the total number of bytes to write
- `off`: byte offset in the object to begin writing at

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

Append *len* bytes from *buf* into the *oid* object. The value of *len* must be <= UINT_MAX/2.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API 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().

Return

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

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

CEPH_RADOS_API int **`rados_checksum`**(rados_ioctx_t *io*, const char * *oid*, rados_checksum_type_t *type*, const char * *init_value*, size_t *init_value_len*, size_t *len*, uint64_t *off*, size_t *chunk_size*, char * *pchecksum*, size_t *checksum_len*)

Compute checksum from object data

The io context determines the snapshot to checksum, if any was set by rados_ioctx_snap_set_read(). The length of the init_value and resulting checksum are dependent upon the checksum type:

XXHASH64: le64 XXHASH32: le32 CRC32C: le32

The checksum result is encoded the following manner:

le32 num_checksum_chunks { leXX checksum for chunk (where XX = appropriate size for the checksum type) } * num_checksum_chunks

Return

negative error code on failure

Parameters

- `io`: the context in which to perform the checksum
- `oid`: the name of the object to checksum
- `type`: the checksum algorithm to utilize
- `init_value`: the init value for the algorithm
- `init_value_len`: the length of the init value
- `len`: the number of bytes to checksum
- `off`: the offset to start checksuming in the object
- `chunk_size`: optional length-aligned chunk size for checksums
- `pchecksum`: where to store the checksum result
- `checksum_len`: the number of bytes available for the result

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

Delete an object

Note

This does not delete any snapshots of the object.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool to delete the object from
- `oid`: the name of the object to delete

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

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

CEPH_RADOS_API int **`rados_cmpext`**(rados_ioctx_t *io*, const char * *o*, const char * *cmp_buf*, size_t *cmp_len*, uint64_t *off*)

Compare an on-disk object range with a buffer

Return

0 on success, negative error code on failure, (-MAX_ERRNO - mismatch_off) on mismatch

Parameters

- `io`: the context in which to perform the comparison
- `o`: name of the object
- `cmp_buf`: buffer containing bytes to be compared with object contents
- `cmp_len`: length to compare and size of in bytes
- `off`: object byte offset at which to start the comparison

Xattrs

Extended attributes are stored as extended attributes on the files representing an object on the OSDs. Thus, they have the same limitations as the underlying filesystem. On ext4, this means that the total data stored in xattrs cannot exceed 4KB.

CEPH_RADOS_API 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.

Return

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

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

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

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

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

Delete an extended attribute from an object.

Return

0 on success, negative error code on failure

Parameters

- `io`: the context in which to delete the xattr
- `o`: the name of the object
- `name`: which xattr to delete

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

Start iterating over xattrs on an object.

Post

iter is a valid iterator

Return

0 on success, negative error code on failure

Parameters

- `io`: the context in which to list xattrs
- `oid`: name of the object

- `iter`: where to store the iterator

CEPH_RADOS_API 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

Pre

iter is a valid iterator

Post

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.

Return

0 on success, negative error code on failure

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

CEPH_RADOS_API 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

Asynchronous Xattrs

Extended attributes are stored as extended attributes on the files representing an object on the OSDs. Thus, they have the same limitations as the underlying filesystem. On ext4, this means that the total data stored in xattrs cannot exceed 4KB.

CEPH_RADOS_API int **`rados_aio_getxattr`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *name*, char * *buf*, size_t *len*)

Asynchronously get the value of an extended attribute on an object.

Return

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

Parameters

- `io`: the context in which the attribute is read
- `o`: name of the object
- `completion`: what to do when the getxattr completes
- `name`: which extended attribute to read
- `buf`: where to store the result
- `len`: size of buf in bytes

CEPH_RADOS_API int **`rados_aio_setxattr`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *name*, const char * *buf*, size_t *len*)

Asynchronously set an extended attribute on an object.

Return

0 on success, negative error code on failure

Parameters

- `io`: the context in which xattr is set
- `o`: name of the object
- `completion`: what to do when the setxattr completes
- `name`: which extended attribute to set
- `buf`: what to store in the xattr
- `len`: the number of bytes in buf

CEPH_RADOS_API int **`rados_aio_rmxattr`**(rados_ioctx_t *io*, const char * *o*, rados_completion_t *completion*, const char * *name*)

Asynchronously delete an extended attribute from an object.

Return

0 on success, negative error code on failure

Parameters

- `io`: the context in which to delete the xattr
- `o`: the name of the object
- `completion`: what to do when the rmxattr completes
- `name`: which xattr to delete

CEPH_RADOS_API int **`rados_aio_getxattrs`**(rados_ioctx_t *io*, const char * *oid*, rados_completion_t *completion*, rados_xattrs_iter_t * *iter*)

Asynchronously start iterating over xattrs on an object.

Post

iter is a valid iterator

Return

0 on success, negative error code on failure

Parameters

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

Hints

CEPH_RADOS_API int **`rados_set_alloc_hint`**(rados_ioctx_t *io*, const char * *o*, uint64_t *expected_object_size*, uint64_t *expected_write_size*)

Set allocation hint for an object

This is an advisory operation, it will always succeed (as if it was submitted with a LIBRADOS_OP_FLAG_FAILOK flag set) and is not guaranteed to do anything on the backend.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the name of the object
- `expected_object_size`: expected size of the object, in bytes
- `expected_write_size`: expected size of writes to the object, in bytes

CEPH_RADOS_API int **`rados_set_alloc_hint2`**(rados_ioctx_t *io*, const char * *o*, uint64_t *expected_object_size*, uint64_t *expected_write_size*, uint32_t *flags*)

Set allocation hint for an object

This is an advisory operation, it will always succeed (as if it was submitted with a LIBRADOS_OP_FLAG_FAILOK flag set) and is not guaranteed to do anything on the backend.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the name of the object
- `expected_object_size`: expected size of the object, in bytes
- `expected_write_size`: expected size of writes to the object, in bytes
- `flags`: hints about future IO patterns

Object Operations

A single rados operation can do multiple operations on one object atomicly. The whole operation will suceed or fail, and no partial results will be visible.

Operations may be either reads, which can return data, or writes, which cannot. The effects of writes are applied and visible all at once, so an operation that sets an xattr and then checks its value will not see the updated value.

CEPH_RADOS_API rados_write_op_t **`rados_create_write_op`**(void)

Create a new rados_write_op_t write operation. This will store all actions to be performed atomically. You must call rados_release_write_op when you are finished with it.

Return

non-NULL on success, NULL on memory allocation error.

CEPH_RADOS_API void **`rados_release_write_op`**(rados_write_op_t *write_op*)

Free a rados_write_op_t, must be called when you're done with it.

Parameters

- `write_op`: operation to deallocate, created with rados_create_write_op

CEPH_RADOS_API void **`rados_write_op_set_flags`**(rados_write_op_t *write_op*, int *flags*)

Set flags for the last operation added to this write_op. At least one op must have been added to the write_op.

Parameters

- `flags`: see librados.h constants beginning with LIBRADOS_OP_FLAG

CEPH_RADOS_API void **`rados_write_op_assert_exists`**(rados_write_op_t *write_op*)

Ensure that the object exists before writing

Parameters

- `write_op`: operation to add this action to

CEPH_RADOS_API void **`rados_write_op_assert_version`**(rados_write_op_t *write_op*, uint64_t *ver*)

Ensure that the object exists and that its internal version number is equal to "ver" before writing. "ver" should be a version number previously obtained with rados_get_last_version().

- If the object's version is greater than the asserted version then rados_write_op_operate will return -ERANGE instead of executing the op.
- If the object's version is less than the asserted version then rados_write_op_operate will return -EOVERFLOW instead of executing the op.

  Parameters

  - `write_op`: operation to add this action to
  - `ver`: object version number

CEPH_RADOS_API void **`rados_write_op_cmpext`**(rados_write_op_t *write_op*, const char * *cmp_buf*, size_t *cmp_len*, uint64_t *off*, int * *prval*)

Ensure that given object range (extent) satisfies comparison.

Parameters

- `write_op`: operation to add this action to
- `cmp_buf`: buffer containing bytes to be compared with object contents
- `cmp_len`: length to compare and size of in bytes
- `off`: object byte offset at which to start the comparison
- `prval`: returned result of comparison, 0 on success, negative error code on failure, (-MAX_ERRNO - mismatch_off) on mismatch

CEPH_RADOS_API void **`rados_write_op_cmpxattr`**(rados_write_op_t *write_op*, const char * *name*, uint8_t *comparison_operator*, const char * *value*, size_t *value_len*)

Ensure that given xattr satisfies comparison. If the comparison is not satisfied, the return code of the operation will be -ECANCELED

Parameters

- `write_op`: operation to add this action to
- `name`: name of the xattr to look up
- `comparison_operator`: currently undocumented, look for LIBRADOS_CMPXATTR_OP_EQ in librados.h
- `value`: buffer to compare actual xattr value to
- `value_len`: length of buffer to compare actual xattr value to

CEPH_RADOS_API void **`rados_write_op_omap_cmp`**(rados_write_op_t *write_op*, const char * *key*, uint8_t *comparison_operator*, const char * *val*, size_t *val_len*, int * *prval*)

Ensure that the an omap value satisfies a comparison, with the supplied value on the right hand side (i.e. for OP_LT, the comparison is actual_value < value.

Parameters

- `write_op`: operation to add this action to
- `key`: which omap value to compare
- `comparison_operator`: one of LIBRADOS_CMPXATTR_OP_EQ, LIBRADOS_CMPXATTR_OP_LT, or LIBRADOS_CMPXATTR_OP_GT
- `val`: value to compare with
- `val_len`: length of value in bytes
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_write_op_setxattr`**(rados_write_op_t *write_op*, const char * *name*, const char * *value*, size_t *value_len*)

Set an xattr

Parameters

- `write_op`: operation to add this action to
- `name`: name of the xattr
- `value`: buffer to set xattr to
- `value_len`: length of buffer to set xattr to

CEPH_RADOS_API void **`rados_write_op_rmxattr`**(rados_write_op_t *write_op*, const char * *name*)

Remove an xattr

Parameters

- `write_op`: operation to add this action to
- `name`: name of the xattr to remove

CEPH_RADOS_API void **`rados_write_op_create`**(rados_write_op_t *write_op*, int *exclusive*, const char * *category*)

Create the object

Parameters

- `write_op`: operation to add this action to
- `exclusive`: set to either LIBRADOS_CREATE_EXCLUSIVE or LIBRADOS_CREATE_IDEMPOTENT will error if the object already exists.
- `category`: category string (DEPRECATED, HAS NO EFFECT)

CEPH_RADOS_API void **`rados_write_op_write`**(rados_write_op_t *write_op*, const char * *buffer*, size_t *len*, uint64_t *offset*)

Write to offset

Parameters

- `write_op`: operation to add this action to
- `offset`: offset to write to
- `buffer`: bytes to write
- `len`: length of buffer

CEPH_RADOS_API void **`rados_write_op_write_full`**(rados_write_op_t *write_op*, const char * *buffer*, size_t *len*)

Write whole object, atomically replacing it.

Parameters

- `write_op`: operation to add this action to
- `buffer`: bytes to write
- `len`: length of buffer

CEPH_RADOS_API void **`rados_write_op_writesame`**(rados_write_op_t *write_op*, const char * *buffer*, size_t *data_len*, size_t *write_len*, uint64_t *offset*)

Write the same buffer multiple times

Parameters

- `write_op`: operation to add this action to
- `buffer`: bytes to write
- `data_len`: length of buffer

- `write_len`: total number of bytes to write, as a multiple of
- `offset`: offset to write to

CEPH_RADOS_API void **`rados_write_op_append`**(rados_write_op_t *write_op*, const char * *buffer*, size_t *len*)

Append to end of object.

Parameters

- `write_op`: operation to add this action to
- `buffer`: bytes to write
- `len`: length of buffer

CEPH_RADOS_API void **`rados_write_op_remove`**(rados_write_op_t *write_op*)

Remove object

Parameters

- `write_op`: operation to add this action to

CEPH_RADOS_API void **`rados_write_op_truncate`**(rados_write_op_t *write_op*, uint64_t *offset*)

Truncate an object

Parameters

- `write_op`: operation to add this action to
- `offset`: Offset to truncate to

CEPH_RADOS_API void **`rados_write_op_zero`**(rados_write_op_t *write_op*, uint64_t *offset*, uint64_t *len*)

Zero part of an object

Parameters

- `write_op`: operation to add this action to
- `offset`: Offset to zero
- `len`: length to zero

CEPH_RADOS_API void **`rados_write_op_exec`**(rados_write_op_t *write_op*, const char * *cls*, const char * *method*, const char * *in_buf*, size_t *in_len*, int * *prval*)

Execute an OSD class method on an object See rados_exec() for general description.

Parameters

- `write_op`: operation to add this action to
- `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
- `prval`: where to store the return value from the method

CEPH_RADOS_API void **`rados_write_op_omap_set`**(rados_write_op_t *write_op*, char const *const * *keys*, char const *const * *vals*, const size_t * *lens*, size_t *num*)

Set key/value pairs on an object

Parameters

- `write_op`: operation to add this action to
- `keys`: array of null-terminated char arrays representing keys to set
- `vals`: array of pointers to values to set
- `lens`: array of lengths corresponding to each value
- `num`: number of key/value pairs to set

CEPH_RADOS_API void **`rados_write_op_omap_rm_keys`**(rados_write_op_t *write_op*, char const *const * *keys*, size_t *keys_len*)

Remove key/value pairs from an object

Parameters

- `write_op`: operation to add this action to
- `keys`: array of null-terminated char arrays representing keys to remove
- `keys_len`: number of key/value pairs to remove

CEPH_RADOS_API void **`rados_write_op_omap_clear`**(rados_write_op_t *write_op*)

Remove all key/value pairs from an object

Parameters

- `write_op`: operation to add this action to

CEPH_RADOS_API void **`rados_write_op_set_alloc_hint`**(rados_write_op_t *write_op*, uint64_t *expected_object_size*, uint64_t *expected_write_size*)

Set allocation hint for an object

Parameters

- `write_op`: operation to add this action to
- `expected_object_size`: expected size of the object, in bytes
- `expected_write_size`: expected size of writes to the object, in bytes

CEPH_RADOS_API void **`rados_write_op_set_alloc_hint2`**(rados_write_op_t *write_op*, uint64_t *expected_object_size*, uint64_t *expected_write_size*, uint32_t *flags*)

Set allocation hint for an object

Parameters

- `write_op`: operation to add this action to
- `expected_object_size`: expected size of the object, in bytes
- `expected_write_size`: expected size of writes to the object, in bytes
- `flags`: hints about future IO patterns

CEPH_RADOS_API int **`rados_write_op_operate`**(rados_write_op_t *write_op*, rados_ioctx_t *io*, const char * *oid*, time_t * *mtime*, int *flags*)

Perform a write operation synchronously

Parameters

- `write_op`: operation to perform
- `io`: the ioctx that the object is in
- `oid`: the object id
- `mtime`: the time to set the mtime to, NULL for the current time
- `flags`: flags to apply to the entire operation (LIBRADOS_OPERATION_*)

CEPH_RADOS_API int **`rados_write_op_operate2`**(rados_write_op_t *write_op*, rados_ioctx_t *io*, const char * *oid*, struct timespec * *mtime*, int *flags*)

Perform a write operation synchronously

Parameters

- `write_op`: operation to perform
- `io`: the ioctx that the object is in
- `oid`: the object id
- `mtime`: the time to set the mtime to, NULL for the current time
- `flags`: flags to apply to the entire operation (LIBRADOS_OPERATION_*)

CEPH_RADOS_API int **`rados_aio_write_op_operate`**(rados_write_op_t *write_op*, rados_ioctx_t *io*, rados_completion_t *completion*, const char * *oid*, time_t * *mtime*, int *flags*)

Perform a write operation asynchronously

Parameters

- `write_op`: operation to perform
- `io`: the ioctx that the object is in
- `completion`: what to do when operation has been attempted
- `oid`: the object id
- `mtime`: the time to set the mtime to, NULL for the current time
- `flags`: flags to apply to the entire operation (LIBRADOS_OPERATION_*)

CEPH_RADOS_API rados_read_op_t **`rados_create_read_op`**(void)

Create a new rados_read_op_t write operation. This will store all actions to be performed atomically. You must call rados_release_read_op when you are finished with it (after it completes, or you decide not to send it in the first place).

Return

non-NULL on success, NULL on memory allocation error.

CEPH_RADOS_API void **`rados_release_read_op`**(rados_read_op_t *read_op*)

Free a rados_read_op_t, must be called when you're done with it.

Parameters

- `read_op`: operation to deallocate, created with rados_create_read_op

CEPH_RADOS_API void **`rados_read_op_set_flags`**(rados_read_op_t *read_op*, int *flags*)

Set flags for the last operation added to this read_op. At least one op must have been added to the read_op.

Parameters

- `flags`: see librados.h constants beginning with LIBRADOS_OP_FLAG

CEPH_RADOS_API void **`rados_read_op_assert_exists`**(rados_read_op_t *read_op*)

Ensure that the object exists before reading

Parameters

- `read_op`: operation to add this action to

CEPH_RADOS_API void **`rados_read_op_assert_version`**(rados_read_op_t *read_op*, uint64_t *ver*)

Ensure that the object exists and that its internal version number is equal to "ver" before reading. "ver" should be a version number previously obtained with rados_get_last_version().

- If the object's version is greater than the asserted version then rados_read_op_operate will return -ERANGE instead of executing the op.
- If the object's version is less than the asserted version then rados_read_op_operate will return -EOVERFLOW instead of executing the op.

  Parameters

  - `read_op`: operation to add this action to
  - `ver`: object version number

CEPH_RADOS_API void **`rados_read_op_cmpext`**(rados_read_op_t *read_op*, const char * *cmp_buf*, size_t *cmp_len*, uint64_t *off*, int * *prval*)

Ensure that given object range (extent) satisfies comparison.

Parameters

- `read_op`: operation to add this action to
- `cmp_buf`: buffer containing bytes to be compared with object contents
- `cmp_len`: length to compare and size of in bytes
- `off`: object byte offset at which to start the comparison
- `prval`: returned result of comparison, 0 on success, negative error code on failure, (-MAX_ERRNO - mismatch_off) on mismatch

CEPH_RADOS_API void **`rados_read_op_cmpxattr`**(rados_read_op_t *read_op*, const char * *name*, uint8_t *comparison_operator*, const char * *value*, size_t *value_len*)

Ensure that the an xattr satisfies a comparison If the comparison is not satisfied, the return code of the operation will be -ECANCELED

Parameters

- `read_op`: operation to add this action to
- `name`: name of the xattr to look up
- `comparison_operator`: currently undocumented, look for LIBRADOS_CMPXATTR_OP_EQ in librados.h
- `value`: buffer to compare actual xattr value to
- `value_len`: length of buffer to compare actual xattr value to

CEPH_RADOS_API void **`rados_read_op_getxattrs`**(rados_read_op_t *read_op*, rados_xattrs_iter_t * *iter*, int * *prval*)

Start iterating over xattrs on an object.

Parameters

- `read_op`: operation to add this action to
- `iter`: where to store the iterator
- `prval`: where to store the return value of this action

CEPH_RADOS_API void **`rados_read_op_omap_cmp`**(rados_read_op_t *read_op*, const char * *key*, uint8_t *comparison_operator*, const char * *val*, size_t *val_len*, int * *prval*)

Ensure that the an omap value satisfies a comparison, with the supplied value on the right hand side (i.e. for OP_LT, the comparison is actual_value < value.

Parameters

- `read_op`: operation to add this action to
- `key`: which omap value to compare
- `comparison_operator`: one of LIBRADOS_CMPXATTR_OP_EQ, LIBRADOS_CMPXATTR_OP_LT, or LIBRADOS_CMPXATTR_OP_GT
- `val`: value to compare with
- `val_len`: length of value in bytes
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_read_op_stat`**(rados_read_op_t *read_op*, uint64_t * *psize*, time_t * *pmtime*, int * *prval*)

Get object size and mtime

Parameters

- `read_op`: operation to add this action to
- `psize`: where to store object size
- `pmtime`: where to store modification time
- `prval`: where to store the return value of this action

CEPH_RADOS_API void **`rados_read_op_read`**(rados_read_op_t *read_op*, uint64_t *offset*, size_t *len*, char * *buffer*, size_t * *bytes_read*, int * *prval*)

Read bytes from offset into buffer.

prlen will be filled with the number of bytes read if successful. A short read can only occur if the read reaches the end of the object.

Parameters

- `read_op`: operation to add this action to
- `offset`: offset to read from
- `len`: length of buffer
- `buffer`: where to put the data
- `bytes_read`: where to store the number of bytes read by this action
- `prval`: where to store the return value of this action

CEPH_RADOS_API void **`rados_read_op_checksum`**(rados_read_op_t *read_op*, rados_checksum_type_t *type*, const char * *init_value*, size_t *init_value_len*, uint64_t *offset*, size_t *len*, size_t *chunk_size*, char * *pchecksum*, size_t *checksum_len*, int * *prval*)

Compute checksum from object data

Parameters

- `read_op`: operation to add this action to
- `oid`: the name of the object to checksum
- `type`: the checksum algorithm to utilize
- `init_value`: the init value for the algorithm
- `init_value_len`: the length of the init value
- `len`: the number of bytes to checksum
- `off`: the offset to start checksuming in the object
- `chunk_size`: optional length-aligned chunk size for checksums
- `pchecksum`: where to store the checksum result for this action
- `checksum_len`: the number of bytes available for the result
- `prval`: where to store the return value for this action

CEPH_RADOS_API void **`rados_read_op_exec`**(rados_read_op_t *read_op*, const char * *cls*, const char * *method*, const char * *in_buf*, size_t *in_len*, char ** *out_buf*, size_t * *out_len*, int * *prval*)

Execute an OSD class method on an object See rados_exec() for general description.

The output buffer is allocated on the heap; the caller is expected to release that memory with rados_buffer_free(). The buffer and length pointers can all be NULL, in which case they are not filled in.

Parameters

- `read_op`: operation to add this action to
- `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
- `out_buf`: where to put librados-allocated output buffer

- `out_len`: length of out_buf in bytes
- `prval`: where to store the return value from the method

CEPH_RADOS_API void **`rados_read_op_exec_user_buf`**(rados_read_op_t *read_op*, const char * *cls*, const char * *method*, const char * *in_buf*, size_t *in_len*, char * *out_buf*, size_t *out_len*, size_t * *used_len*, int * *prval*)

Execute an OSD class method on an object See rados_exec() for general description.

If the output buffer is too small, prval will be set to -ERANGE and used_len will be 0.

Parameters

- `read_op`: operation to add this action to
- `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
- `out_buf`: user-provided buffer to read into
- `out_len`: length of out_buf in bytes
- `used_len`: where to store the number of bytes read into out_buf
- `prval`: where to store the return value from the method

CEPH_RADOS_API void **`rados_read_op_omap_get_vals`**(rados_read_op_t *read_op*, const char * *start_after*, const char * *filter_prefix*, uint64_t *max_return*, rados_omap_iter_t * *iter*, int * *prval*)

Start iterating over key/value pairs on an object.

They will be returned sorted by key.

Parameters

- `read_op`: operation to add this action to
- `start_after`: list keys starting after start_after
- `filter_prefix`: list only keys beginning with filter_prefix
- `max_return`: list no more than max_return key/value pairs
- `iter`: where to store the iterator
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_read_op_omap_get_vals2`**(rados_read_op_t *read_op*, const char * *start_after*, const char * *filter_prefix*, uint64_t *max_return*, rados_omap_iter_t * *iter*, unsigned char * *pmore*, int * *prval*)

Start iterating over key/value pairs on an object.

They will be returned sorted by key.

Parameters

- `read_op`: operation to add this action to
- `start_after`: list keys starting after start_after
- `filter_prefix`: list only keys beginning with filter_prefix
- `max_return`: list no more than max_return key/value pairs
- `iter`: where to store the iterator
- `pmore`: flag indicating whether there are more keys to fetch
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_read_op_omap_get_keys`**(rados_read_op_t *read_op*, const char * *start_after*, uint64_t *max_return*, rados_omap_iter_t * *iter*, int * *prval*)

Start iterating over keys on an object.

They will be returned sorted by key, and the iterator will fill in NULL for all values if specified.

Parameters

- `read_op`: operation to add this action to
- `start_after`: list keys starting after start_after
- `max_return`: list no more than max_return keys
- `iter`: where to store the iterator
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_read_op_omap_get_keys2`**(rados_read_op_t *read_op*, const char * *start_after*, uint64_t *max_return*, rados_omap_iter_t * *iter*, unsigned char * *pmore*, int * *prval*)

Start iterating over keys on an object.

They will be returned sorted by key, and the iterator will fill in NULL for all values if specified.

Parameters

- `read_op`: operation to add this action to
- `start_after`: list keys starting after start_after
- `max_return`: list no more than max_return keys
- `iter`: where to store the iterator
- `pmore`: flag indicating whether there are more keys to fetch
- `prval`: where to store the return value from this action

CEPH_RADOS_API void **`rados_read_op_omap_get_vals_by_keys`**(rados_read_op_t *read_op*, char const *const * *keys*, size_t *keys_len*, rados_omap_iter_t * *iter*, int * *prval*)

Start iterating over specific key/value pairs

They will be returned sorted by key.

Parameters

- `read_op`: operation to add this action to
- `keys`: array of pointers to null-terminated keys to get
- `keys_len`: the number of strings in keys
- `iter`: where to store the iterator
- `prval`: where to store the return value from this action

CEPH_RADOS_API int **`rados_read_op_operate`**(rados_read_op_t *read_op*, rados_ioctx_t *io*, const char * *oid*, int *flags*)

Perform a read operation synchronously

Parameters

- `read_op`: operation to perform
- `io`: the ioctx that the object is in
- `oid`: the object id
- `flags`: flags to apply to the entire operation (LIBRADOS_OPERATION_*)

CEPH_RADOS_API int **`rados_aio_read_op_operate`**(rados_read_op_t *read_op*, rados_ioctx_t *io*, rados_completion_t *completion*, const char * *oid*, int *flags*)

Perform a read operation asynchronously

Parameters

- `read_op`: operation to perform
- `io`: the ioctx that the object is in
- `completion`: what to do when operation has been attempted
- `oid`: the object id
- `flags`: flags to apply to the entire operation (LIBRADOS_OPERATION_*)

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`(maj, min, extra)

`LIBRADOS_VERSION_CODE`

`LIBRADOS_SUPPORTS_WATCH`

`LIBRADOS_SUPPORTS_SERVICES`

`LIBRADOS_SUPPORTS_APP_METADATA`

`LIBRADOS_LOCK_FLAG_RENEW`

`LIBRADOS_CREATE_EXCLUSIVE`

`LIBRADOS_CREATE_IDEMPOTENT`

`CEPH_RADOS_API`

`LIBRADOS_SNAP_HEAD`

`LIBRADOS_SNAP_DIR`

`VOIDPTR_RADOS_T`

Typedefs

typedef **`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 or to the same cluster with different users requires different cluster handles.

typedef **`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.

typedef **`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())
- namespace for all single-object operations (see rados_ioctx_set_namespace()). Set to LIBRADOS_ALL_NSPACES before rados_nobjects_list_open() will list all objects in all namespaces.

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

typedef **`rados_list_ctx_t`**

An iterator for listing the objects in a pool. Used with rados_nobjects_list_open(), rados_nobjects_list_next(), and rados_nobjects_list_close().

typedef **`rados_object_list_cursor`**

The cursor used with rados_enumerate_objects and accompanying methods.

typedef struct rados_object_list_item **`rados_object_list_item`**

typedef **`rados_snap_t`**

The id of a snapshot.

typedef **`rados_xattrs_iter_t`**

An iterator for listing extended attrbutes on an object. Used with rados_getxattrs(), rados_getxattrs_next(), and rados_getxattrs_end().

typedef **`rados_omap_iter_t`**

An iterator for listing omap key/value pairs on an object. Used with rados_read_op_omap_get_keys(), rados_read_op_omap_get_vals(), rados_read_op_omap_get_vals_by_keys(), rados_omap_get_next(), and rados_omap_get_end().

typedef **`rados_write_op_t`**

An object write operation stores a number of operations which can be executed atomically. For usage, see:

- Creation and deletion: rados_create_write_op() rados_release_write_op()
- Extended attribute manipulation: rados_write_op_cmpxattr() rados_write_op_cmpxattr(), rados_write_op_setxattr(), rados_write_op_rmxattr()
- Object map key/value pairs: rados_write_op_omap_set(), rados_write_op_omap_rm_keys(), rados_write_op_omap_clear(), rados_write_op_omap_cmp()
- Object properties: rados_write_op_assert_exists(), rados_write_op_assert_version()
- Creating objects: rados_write_op_create()
- IO on objects: rados_write_op_append(), rados_write_op_write(), rados_write_op_zero rados_write_op_write_full(), rados_write_op_writesame(), rados_write_op_remove, rados_write_op_truncate(), rados_write_op_zero(), rados_write_op_cmpext()
- Hints: rados_write_op_set_alloc_hint()
- Performing the operation: rados_write_op_operate(), rados_aio_write_op_operate()

typedef **`rados_read_op_t`**

An object read operation stores a number of operations which can be executed atomically. For usage, see:

- Creation and deletion: rados_create_read_op() rados_release_read_op()
- Extended attribute manipulation: rados_read_op_cmpxattr(), rados_read_op_getxattr(), rados_read_op_getxattrs()
- Object map key/value pairs: rados_read_op_omap_get_vals(), rados_read_op_omap_get_keys(), rados_read_op_omap_get_vals_by_keys(), rados_read_op_omap_cmp()
- Object properties: rados_read_op_stat(), rados_read_op_assert_exists(), rados_read_op_assert_version()
- IO on objects: rados_read_op_read(), rados_read_op_checksum(), rados_read_op_cmpext()
- Custom operations: rados_read_op_exec(), rados_read_op_exec_user_buf()
- Request properties: rados_read_op_set_flags()
- Performing the operation: rados_read_op_operate(), rados_aio_read_op_operate()

typedef **`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.

Enums

enum **`[anonymous]`**

Values:

`LIBRADOS_OP_FLAG_EXCL` = 0x1

`LIBRADOS_OP_FLAG_FAILOK` = 0x2

`LIBRADOS_OP_FLAG_FADVISE_RANDOM` = 0x4

`LIBRADOS_OP_FLAG_FADVISE_SEQUENTIAL` = 0x8

`LIBRADOS_OP_FLAG_FADVISE_WILLNEED` = 0x10

`LIBRADOS_OP_FLAG_FADVISE_DONTNEED` = 0x20

`LIBRADOS_OP_FLAG_FADVISE_NOCACHE` = 0x40

enum **`rados_checksum_type_t`**

Values:

`LIBRADOS_CHECKSUM_TYPE_XXHASH32` = 0

`LIBRADOS_CHECKSUM_TYPE_XXHASH64` = 1

`LIBRADOS_CHECKSUM_TYPE_CRC32C` = 2

Functions

CEPH_RADOS_API 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

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster to query
- `result`: where to store the results

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Return

-ERANGE if the buffer is too short to contain the fsid

Parameters

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

CEPH_RADOS_API int **`rados_wait_for_latest_osdmap`**(rados_t *cluster*)

Get/wait for the most recent osdmap

Return

0 on sucess, negative error code on failure

Parameters

- `cluster`: the cluster to shutdown

CEPH_RADOS_API int **`rados_omap_get_next`**(rados_omap_iter_t *iter*, char ** *key*, char ** *val*, size_t * *len*)

Get the next omap key/value pair on the object

Pre

iter is a valid iterator

Post

key and val are the next key/value pair. key is null-terminated, and val has length len. If the end of the list has been reached, key and val are NULL, and len is 0. key and val will not be accessible after rados_omap_get_end() is called on iter, so if they are needed after that they should be copied.

Return

0 on success, negative error code on failure

Parameters

- `iter`: iterator to advance
- `key`: where to store the key of the next omap entry
- `val`: where to store the value of the next omap entry
- `len`: where to store the number of bytes in val

CEPH_RADOS_API void **`rados_omap_get_end`**(rados_omap_iter_t *iter*)

Close the omap iterator.

iter should not be used after this is called.

Parameters

- `iter`: the iterator to close

CEPH_RADOS_API 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?

Return

0 on success, negative error code on failure

Parameters

- `io`: ioctx
- `o`: object name
- `psize`: where to store object size
- `pmtime`: where to store modification time

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Parameters

- `io`: ioctx
- `o`: object name
- `cmdbuf`: command buffer
- `cmdbuflen`: command buffer length in bytes

CEPH_RADOS_API 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

Return

0 on success, negative error code on failure

Parameters

- `io`: ioctx
- `o`: object name
- `buf`: buffer
- `buflen`: buffer length in bytes

CEPH_RADOS_API 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.

Return

0 on success, negative error code on failure

Return

-ERANGE if buf isn't big enough

Parameters

- `io`: ioctx
- `o`: object name
- `buf`: buffer
- `buflen`: buffer length in bytes

CEPH_RADOS_API 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 subdirectories

Return

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.

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

CEPH_RADOS_API int **`rados_cache_pin`**(rados_ioctx_t *io*, const char * *o*)

Pin an object in the cache tier

When an object is pinned in the cache tier, it stays in the cache tier, and won't be flushed out.

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the object id

CEPH_RADOS_API int **`rados_cache_unpin`**(rados_ioctx_t *io*, const char * *o*)

Unpin an object in the cache tier

After an object is unpinned in the cache tier, it can be flushed out

Return

0 on success, negative error code on failure

Parameters

- `io`: the pool the object is in
- `o`: the object id

CEPH_RADOS_API int **`rados_lock_exclusive`**(rados_ioctx_t *io*, const char * *oid*, const char * *name*, const char * *cookie*, const char * *desc*, struct timeval * *duration*, uint8_t *flags*)

Take an exclusive lock on an object.

Return

0 on success, negative error code on failure

Return

-EBUSY if the lock is already held by another (client, cookie) pair

Return

-EEXIST if the lock is already held by the same (client, cookie) pair

Parameters

- `io`: the context to operate in
- `oid`: the name of the object
- `name`: the name of the lock
- `cookie`: user-defined identifier for this instance of the lock
- `desc`: user-defined lock description
- `duration`: the duration of the lock. Set to NULL for infinite duration.
- `flags`: lock flags

CEPH_RADOS_API int **`rados_lock_shared`**(rados_ioctx_t *io*, const char * *o*, const char * *name*, const char * *cookie*, const char * *tag*, const char * *desc*, struct timeval * *duration*, uint8_t *flags*)

Take a shared lock on an object.

Return

0 on success, negative error code on failure

Return

-EBUSY if the lock is already held by another (client, cookie) pair

Return

-EEXIST if the lock is already held by the same (client, cookie) pair

Parameters

- `io`: the context to operate in
- `o`: the name of the object
- `name`: the name of the lock
- `cookie`: user-defined identifier for this instance of the lock
- `tag`: The tag of the lock
- `desc`: user-defined lock description
- `duration`: the duration of the lock. Set to NULL for infinite duration.
- `flags`: lock flags

CEPH_RADOS_API int **`rados_unlock`**(rados_ioctx_t *io*, const char * *o*, const char * *name*, const char * *cookie*)

Release a shared or exclusive lock on an object.

Return

0 on success, negative error code on failure

Return

-ENOENT if the lock is not held by the specified (client, cookie) pair

Parameters

- `io`: the context to operate in
- `o`: the name of the object
- `name`: the name of the lock
- `cookie`: user-defined identifier for the instance of the lock

CEPH_RADOS_API int **`rados_aio_unlock`**(rados_ioctx_t *io*, const char * *o*, const char * *name*, const char * *cookie*, rados_completion_t *completion*)

Asynchronous release a shared or exclusive lock on an object.

Return

0 on success, negative error code on failure

Parameters

- `io`: the context to operate in
- `o`: the name of the object
- `name`: the name of the lock
- `cookie`: user-defined identifier for the instance of the lock
- `completion`: what to do when operation has been attempted

CEPH_RADOS_API ssize_t **`rados_list_lockers`**(rados_ioctx_t *io*, const char * *o*, const char * *name*, int * *exclusive*, char * *tag*, size_t * *tag_len*, char * *clients*, size_t * *clients_len*, char * *cookies*, size_t * *cookies_len*, char * *addrs*, size_t * *addrs_len*)

List clients that have locked the named object lock and information about the lock.

The number of bytes required in each buffer is put in the corresponding size out parameter. If any of the provided buffers are too short, -ERANGE is returned after these sizes are filled in.

Return

number of lockers on success, negative error code on failure

Return

-ERANGE if any of the buffers are too short

Parameters

- `io`: the context to operate in
- `o`: the name of the object
- `name`: the name of the lock
- `exclusive`: where to store whether the lock is exclusive (1) or shared (0)
- `tag`: where to store the tag associated with the object lock
- `tag_len`: number of bytes in tag buffer
- `clients`: buffer in which locker clients are stored, separated by '\0'
- `clients_len`: number of bytes in the clients buffer
- `cookies`: buffer in which locker cookies are stored, separated by '\0'
- `cookies_len`: number of bytes in the cookies buffer
- `addrs`: buffer in which locker addresses are stored, separated by '\0'
- `addrs_len`: number of bytes in the clients buffer

CEPH_RADOS_API int **`rados_break_lock`**(rados_ioctx_t *io*, const char * *o*, const char * *name*, const char * *client*, const char * *cookie*)

Releases a shared or exclusive lock on an object, which was taken by the specified client.

Return

0 on success, negative error code on failure

Return

-ENOENT if the lock is not held by the specified (client, cookie) pair

Return

-EINVAL if the client cannot be parsed

Parameters

- `io`: the context to operate in
- `o`: the name of the object
- `name`: the name of the lock

- `client`: the client currently holding the lock
- `cookie`: user-defined identifier for the instance of the lock

CEPH_RADOS_API int **`rados_blacklist_add`**(rados_t *cluster*, char * *client_address*, uint32_t *expire_seconds*)

Blacklists the specified client from the OSDs

Return

0 on success, negative error code on failure

Parameters

- `cluster`: cluster handle
- `client_address`: client address
- `expire_seconds`: number of seconds to blacklist (0 for default)

CEPH_RADOS_API void **`rados_set_osdmap_full_try`**(rados_ioctx_t *io*)

CEPH_RADOS_API void **`rados_unset_osdmap_full_try`**(rados_ioctx_t *io*)

CEPH_RADOS_API int **`rados_application_enable`**(rados_ioctx_t *io*, const char * *app_name*, int *force*)

Enable an application on a pool

Return

0 on success, negative error code on failure

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `force`: 0 if only single application per pool

CEPH_RADOS_API int **`rados_application_list`**(rados_ioctx_t *io*, char * *values*, size_t * *values_len*)

List all enabled applications

If the provided buffer is too short, the required length is filled in and -ERANGE is returned. Otherwise, the buffers are filled with the application names, with a '\0' after each.

Return

0 on success, negative error code on failure

Return

-ERANGE if either buffer is too short

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `values`: buffer in which to store application names
- `vals_len`: number of bytes in values buffer

CEPH_RADOS_API int **`rados_application_metadata_get`**(rados_ioctx_t *io*, const char * *app_name*, const char * *key*, char * *value*, size_t * *value_len*)

Get application metadata value from pool

Return

0 on success, negative error code on failure

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `key`: metadata key
- `value`: result buffer
- `value_len`: maximum len of value

CEPH_RADOS_API int **`rados_application_metadata_set`**(rados_ioctx_t *io*, const char * *app_name*, const char * *key*, const char * *value*)

Set application metadata on a pool

Return

0 on success, negative error code on failure

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `key`: metadata key
- `value`: metadata key

CEPH_RADOS_API int **`rados_application_metadata_remove`**(rados_ioctx_t *io*, const char * *app_name*, const char * *key*)

Remove application metadata from a pool

Return

0 on success, negative error code on failure

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `key`: metadata key

CEPH_RADOS_API int **`rados_application_metadata_list`**(rados_ioctx_t *io*, const char * *app_name*, char * *keys*, size_t * *key_len*, char * *values*, size_t * *vals_len*)

List all metadata key/value pairs associated with an application.

This iterates over all metadata, key_len and val_len are filled in with the number of bytes put into the keys and values buffers.

If the provided buffers are too short, the required lengths are filled in and -ERANGE is returned. Otherwise, the buffers are filled with the keys and values of the metadata, with a '\0' after each.

Return

0 on succcess, negative error code on failure

Return

-ERANGE if either buffer is too short

Parameters

- `ioctx`: pool ioctx
- `app_name`: application name
- `keys`: buffer in which to store key names
- `keys_len`: number of bytes in keys buffer
- `values`: buffer in which to store values
- `vals_len`: number of bytes in values buffer

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

CEPH_RADOS_API uint32_t **`rados_objects_list_get_pg_hash_position`**(rados_list_ctx_t *ctx*)

CEPH_RADOS_API uint32_t **`rados_objects_list_seek`**(rados_list_ctx_t *ctx*, uint32_t *pos*)

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

CEPH_RADOS_API void **`rados_objects_list_close`**(rados_list_ctx_t *ctx*)

struct **`rados_object_list_item`**

Public Members

size_t **`oid_length`**

char* **`oid`**

size_t **`nspace_length`**

char* **`nspace`**

size_t **`locator_length`**

char* **`locator`**

struct **`rados_pool_stat_t`**

#include <librados.h>

Usage information for a pool.

Public 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`**

#include <librados.h>

Cluster-wide usage information

Public Members

uint64_t **`kb`**

uint64_t **`kb_used`**

uint64_t **`kb_avail`**

uint64_t **`num_objects`**