RADOS CLIENT PROTOCOL

This is very incomplete, but one must start somewhere.

BASICS

Requests are MOSDOp messages. Replies are MOSDOpReply messages.

An object request is targetted at an hobject_t, which includes a pool, hash value, object name, placement key (usually empty), and snapid.

The hash value is a 32-bit hash value, normally generated by hashing the object name. The hobject_t can be arbitrarily constructed, though, with any hash value and name. Note that in the MOSDOp these components are spread across several fields and not logically assembled in an actual hobject_t member (mainly historical reasons).

A request can also target a PG. In this case, the *ps* value matches a specific PG, the object name is empty, and (hopefully) the ops in the request are PG ops.

Either way, the request ultimately targets a PG, either by using the explicit pgid or by folding the hash value onto the current number of pgs in the pool. The client sends the request to the primary for the assocated PG.

Each request is assigned a unique tid.

RESENDS

If there is a connection drop, the client will resend any outstanding requets.

Any time there is a PG mapping change such that the primary changes, the client is responsible for resending the request. Note that although there may be an interval change from the OSD's perspective (triggering PG peering), if the primary doesn't change then the client need not resend.

There are a few exceptions to this rule:

- There is a last_force_op_resend field in the pg_pool_t in the OSDMap. If this changes, then the clients are forced to resend any outstanding requests. (This happens when tiering is adjusted, for example.)
- Some requests are such that they are resent on *any* PG interval change, as defined by pg_interval_t's is_new_interval() (the same criteria used by peering in the OSD).
- If the PAUSE OSDMap flag is set and unset.

Each time a request is sent to the OSD the *attempt* field is incremented. The first time it is 0, the next 1, etc.

BACKOFF

Ordinarily the OSD will simply queue any requests it can't immeidately process in memory until such time as it can. This can become problematic because the OSD limits the total amount of RAM consumed by incoming messages: if either of the thresholds for the number of messages or the number of bytes is reached, new messages will not be read off the network socket, causing backpressure through the network.

In some cases, though, the OSD knows or expects that a PG or object will be unavailable for some time and does not want to consume memory by queuing requests. In these cases it can send a MOSDBackoff message to the client.

A backoff request has four properties:

1. the op code (block, unblock, or ack-block)
2. *id*, a unique id assigned within this session
3. hobject_t begin
4. hobject_t end

There are two types of backoff: a *PG* backoff will plug all requests targetting an entire PG at the client, as described by a range of the hash/hobject_t space [begin,end), while an *object* backoff will plug all requests targetting a single object (begin == end).

When the client receives a *block* backoff message, it is now responsible for *not* sending any requests for hobject_ts described by the backoff. The backoff remains in effect until the backoff is cleared (via an 'unblock' message) or the OSD session is closed. A *ack_block* message is sent back to the OSD immediately to acknowledge receipt of the backoff.

When an unblock is received, it will reference a specific id that the client previous had blocked. However, the range described by the unblock may be smaller than the original range, as the PG may have split on the OSD. The unblock should *only* unblock the range specified in the unblock message. Any requests that fall within the unblock request range are reexamined and, if no other installed backoff applies, resent.

On the OSD, Backoffs are also tracked across ranges of the hash space, and exist in three states:

1. new
2. acked
3. deleting

A newly installed backoff is set to *new* and a message is sent to the client. When the *ack-block* message is received it is changed to the *acked* state. The OSD may process other messages from the client that are covered by the backoff in the *new* state, but once the backoff is *acked* it should never see a blocked request unless there is a bug.

If the OSD wants to a remove a backoff in the *acked* state it can simply remove it and notify the client. If the backoff is in the *new* state it must move it to the *deleting* state and continue to use it to discard client requests until the *ack-block* message is received, at which point it can finally be removed. This is necessary to preserve the order of operations processed by the OSD.