MAP AND PG MESSAGE HANDLING

OVERVIEW

The OSD handles routing incoming messages to PGs, creating the PG if necessary in some cases.

PG messages generally come in two varieties:

1. Peering Messages
2. Ops/SubOps

There are several ways in which a message might be dropped or delayed. It is important that the message delaying does not result in a violation of certain message ordering requirements on the way to the relevant PG handling logic:

1. Ops referring to the same object must not be reordered.
2. Peering messages must not be reordered.
3. Subops must not be reordered.

MOSDMAP

MOSDMap messages may come from either monitors or other OSDs. Upon receipt, the OSD must perform several tasks:

1. Persist the new maps to the filestore. Several PG operations rely on having access to maps dating back to the last time the PG was clean.
2. Update and persist the superblock.
3. Update OSD state related to the current map.
4. Expose new maps to PG processes via *OSDService*.
5. Remove PGs due to pool removal.
6. Queue dummy events to trigger PG map catchup.

Each PG asynchronously catches up to the currently published map during process_peering_events before processing the event. As a result, different PGs may have different views as to the “current” map.

One consequence of this design is that messages containing submessages from multiple PGs (MOSDPGInfo, MOSDPGQuery, MOSDPGNotify) must tag each submessage with the PG’s epoch as well as tagging the message as a whole with the OSD’s current published epoch.

MOSDPGOP/MOSDPGSUBOP

See OSD::dispatch_op, OSD::handle_op, OSD::handle_sub_op

MOSDPGOps are used by clients to initiate rados operations. MOSDSubOps are used between OSDs to coordinate most non peering activities including replicating MOSDPGOp operations.

OSD::require_same_or_newer map checks that the current OSDMap is at least as new as the map epoch indicated on the message. If not, the message is queued in OSD::waiting_for_osdmap via OSD::wait_for_new_map. Note, this cannot violate the above conditions since any two messages will be queued in order of receipt and if a message is recieved with epoch e0, a later message from the same source must be at epoch at least e0. Note that two PGs from the same OSD count for these purposes as different sources for single PG messages. That is, messages from different PGs may be reordered.

MOSDPGOps follow the following process:

1. OSD::handle_op: validates permissions and crush mapping. discard the request if they are not connected and the client cannot get the reply (See OSD::op_is_discardable) See OSDService::handle_misdirected_op See OSD::op_has_sufficient_caps See OSD::require_same_or_newer_map
2. OSD::enqueue_op

MOSDSubOps follow the following process:

1. OSD::handle_sub_op checks that sender is an OSD
2. OSD::enqueue_op

OSD::enqueue_op calls PG::queue_op which checks waiting_for_map before calling OpWQ::queue which adds the op to the queue of the PG responsible for handling it.

OSD::dequeue_op is then eventually called, with a lock on the PG. At this time, the op is passed to PG::do_request, which checks that:

1. the PG map is new enough (PG::must_delay_op)
2. the client requesting the op has enough permissions (PG::op_has_sufficient_caps)
3. the op is not to be discarded (PG::can_discard_{request,op,subop,scan,backfill})
4. the PG is active (PG::flushed boolean)
5. the op is a CEPH_MSG_OSD_OP and the PG is in PG_STATE_ACTIVE state and not in PG_STATE_REPLAY

If these conditions are not met, the op is either discarded or queued for later processing. If all conditions are met, the op is processed according to its type:

1. CEPH_MSG_OSD_OP is handled by PG::do_op
2. MSG_OSD_SUBOP is handled by PG::do_sub_op
3. MSG_OSD_SUBOPREPLY is handled by PG::do_sub_op_reply
4. MSG_OSD_PG_SCAN is handled by PG::do_scan
5. MSG_OSD_PG_BACKFILL is handled by PG::do_backfill

CEPH_MSG_OSD_OP PROCESSING

ReplicatedPG::do_op handles CEPH_MSG_OSD_OP op and will queue it

1. in wait_for_all_missing if it is a CEPH_OSD_OP_PGLS for a designated snapid and some object updates are still missing
2. in waiting_for_active if the op may write but the scrubber is working
3. in waiting_for_missing_object if the op requires an object or a snapdir or a specific snap that is still missing
4. in waiting_for_degraded_object if the op may write an object or a snapdir that is degraded, or if another object blocks it (“blocked_by”)
5. in waiting_for_backfill_pos if the op requires an object that will be available after the backfill is complete
6. in waiting_for_ack if an ack from another OSD is expected
7. in waiting_for_ondisk if the op is waiting for a write to complete

PEERING MESSAGES

See OSD::handle_pg_(notify|info|log|query)

Peering messages are tagged with two epochs:

1. epoch_sent: map epoch at which the message was sent
2. query_epoch: map epoch at which the message triggering the message was sent

These are the same in cases where there was no triggering message. We discard a peering message if the message’s query_epoch if the PG in question has entered a new epoch (See PG::old_peering_event, PG::queue_peering_event). Notifies, infos, notifies, and logs are all handled as PG::RecoveryMachine events and are wrapped by PG::queue_* by PG::CephPeeringEvts, which include the created state machine event along with epoch_sent and query_epoch in order to generically check PG::old_peering_message upon insertion and removal from the queue.

Note, notifies, logs, and infos can trigger the creation of a PG. See OSD::get_or_create_pg.