MAP AND PG MESSAGE HANDLING

OVERVIEW

The OSD handles routing incoming messages to PGs, creating the PG if necessary in come 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. 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 can_discard_request before queueing the op in the op_queue and the PG in the OpWQ. Note, a single PG may be in the op queue multiple times for multiple ops.

dequeue_op is then eventually called on the PG. At this time, the op is popped off of op_queue and passed to PG::do_request, which checks that the PG map is new enough (must_delay_op) and then processes the request.

In summary, the possible ways that an op may wait or be discarded in are:

1. Wait in waiting_for_osdmap due to OSD::require_same_or_newer_map from OSD::handle_*.
2. Discarded in OSD::can_discard_op at enqueue_op.
3. Wait in PG::op_waiters due to PG::must_delay_request in PG::do_request.
4. Wait in PG::waiting_for_active in due_request due to !flushed.
5. Wait in PG::waiting_for_active due to !active() in do_op/do_sub_op.
6. Wait in PG::waiting_for_(degraded|missing) in do_op.
7. Wait in PG::waiting_for_active due to scrub_block_writes in do_op

TODO: The above is not a complete list.

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.