

WATCH NOTIFY

See librados for the watch/notify interface.

OVERVIEW

The `object_info` (See `osd/osd_types.h`) tracks the set of watchers for a particular object persistently in the `object_info_t::watchers` map. In order to track notify progress, we also maintain some ephemeral structures associated with the `ObjectContext`.

Each Watch has an associated Watch object (See `osd/Watch.h`). The `ObjectContext` for a watched object will have a (strong) reference to one Watch object per watch, and each Watch object holds a reference to the corresponding `ObjectContext`. This circular reference is deliberate and is broken when the Watch state is discarded on a new peering interval or removed upon timeout expiration or an unwatch operation.

A watch tracks the associated connection via a strong `ConnectionRef` `Watch::conn`. The associated connection has a `WatchConState` stashed in the `OSD::Session` for tracking associated Watches in order to be able to notify them upon `ms_handle_reset()` (via `WatchConState::reset()`).

Each Watch object tracks the set of currently un-acked notifies. `start_notify()` on a Watch object adds a reference to a new in-progress Notify to the Watch and either:

- if the Watch is *connected*, sends a Notify message to the client
- if the Watch is *unconnected*, does nothing.

When the Watch becomes connected (in `ReplicatedPG::do_osd_op_effects`), Notifies are resent to all remaining tracked Notify objects.

Each Notify object tracks the set of un-notified Watchers via calls to `complete_watcher()`. Once the remaining set is empty or the timeout expires (cb, registered in `init()`) a notify completion is sent to the client.

WATCH LIFECYCLE

A watch may be in one of 5 states:

1. Non existent.
2. On disk, but not registered with an object context.
3. Connected
4. Disconnected, callback registered with timer
5. Disconnected, callback in queue for scrub or `is_degraded`

Case 2 occurs between when an OSD goes active and the `ObjectContext` for an object with watchers is loaded into memory due to an access. During Case 2, no state is registered for the watch. Case 2 transitions to Case 4 in `ReplicatedPG::populate_obc_watchers()` during `ReplicatedPG::find_object_context`. Case 1 becomes case 3 via `OSD::do_osd_op_effects` due to a watch operation. Case 4,5 become case 3 in the same way. Case 3 becomes case 4 when the connection resets on a watcher's session.

Cases 4&5 can use some explanation. Normally, when a Watch enters Case 4, a callback is registered with the `OSDService::watch_timer` to be called at timeout expiration. At the time that the callback is called, however, the pg might be in a state where it cannot write to the object in order to remove the watch (i.e., during a scrub or while the object is degraded). In that case, we use `Watch::get_delayed_cb()` to generate another Context for use from the `callbacks_for_degraded_object` and `Scrubber::callbacks` lists. In either case, `Watch::unregister_cb()` does the right thing (`SafeTimer::cancel_event()` is harmless for contexts not registered with the timer).

NOTIFY LIFECYCLE

The notify timeout is simpler: a timeout callback is registered when the notify is `init()`'d. If all watchers ack notifies before the timeout occurs, the timeout is canceled and the client is notified of the notify completion. Otherwise, the timeout fires, the Notify object pings each Watch via `cancel_notify` to remove itself, and sends the notify completion to the client early.