Ceph RADOS level replication

Xuehan Xu

2017-07-20

Summary

Recently, after experiencing a series of disasters like man-made misoperations, network problems, we find the real-time cross-cluster replication mechanism an ugent need for our ceph storage system. This demand not only lies in our RBD services, but also rgw and cephfs. As far as I know, both RGW and RBD have their own cross-cluster replication mechanism, while cephfs does not. And on the other hand, we think that if such a mechanism can be implemented at the RADOS layer, it would save a lot of work of current and future upper level systems.

Besides the convenience for the upper level systems, we think that implementation of such a mechanism at the RADOS layer could, in itself, have some advantages over the implementation at upper layers like RBD snd RGW. For example, instead of first saving "journal objects", like in rbd mirroring, to the underlying RADOS system and reading them out and replicate the corresponding operation to another ceph cluster, RADOS level replication can directly replicate "repop"s asynchronously, saving the Disk IO and lock contentions with the RADOS's "normal" I/O processing.

Based on the above consideration, we designed a RADOS level replication function.

Owners

Xuehan Xu (Qihoo 360)
Shuguang Wang (Qihoo 360)
Yufang Zhang (Qihoo 360)
Yupeng Chen (Qihoo 360)
Zhongyan Gu (Qihoo 360)
Huilin Fang (Qihoo 360)
Kun Feng (Qihoo 360)

Current Status

There is still no RADOS level repliation mechanism for Ceph now.

Detailed Description

Design Principle

To implement such a function, the easiest way is to reuse ceph's inter-OSD repop replication mechanism, with the only difference that repops are replicated asynchronously this time. However, doing so involves some other issues as shown below:

How to deal with OSD failure?
: The procedure of "Peering + recovery + backfilling" can insure data consistency within a single cluster, but it doesn't insure "journal" consistency. By "journal consistency", we mean that, in the presence of OSD failure, journal neccesary for the replication of repops that are not replicated before the failure is not lost. And, since the only thing we can reply on when there is OSD failure is the OSD journal, insuring "journal" consistency is essential to our replication function.

How to insure consistency for cross object Ops?
: Since upper level system like RBD may issue an OP that cross multiple objects, insuring cross-object consistency is also essential for the replication function to work correctly.

On the other hand, we find that, although Peering only insures data consistency, if there is infinite disk space to store the journal and OSD never delete journal, Peering can also insure journal consistency, since data are cumulated by the contents of journal. Of course, we can't store infinite journal, but we don't need that much either, we only need the journal that corresponds to not replicated repops. In addition, we can basically clarify journal into two categories: "original op journal" and "recovery journal". The former is the journal that corresponds to client op, while the latter corresponds to osd recovery op. It's easy to see that data stored in RADOS can be cumulated with "original op journal" only. So "original op journal" is what we concerned. To solve the first issue, there is another problem: how to make sure the correctness during "recovery" phase. This can be done by make sure that every object's recovery source replicate all journal related to the recovering object before starts pushing it. So, generally speaking, to solve the first issue, we make sure "original op journal" gets removed only after its corresponding "original op" is replicated and, in the recovery/backfill phase, recovery source replicate all journal related to the recovering object before pushing.

To solve the second issue, we introduce the concept "object set" to adapt to the corresponding concepts of the upper level system, like rbd image in RBD. Each object set is associated with an "obj_set_id" that uniquely identify the object set. Each upper level system OP is associate with a sequenctial, monotonically increasing id "obj_set_op_seq" and a set "obj_set_extents" which indicates which object is involved in this OP. When replicating, OSDs send repops with the same "obj_set_id" and "obj_set_op_seq" to the same intermediate node which, only when all repops within a "obj_set_extents" arrived, sends these repops to the other cluster on behalf of the current cluster. In addition, intermediate nodes respond osds in current cluster with a SUCCESS only when all repops within a "obj_set_extents" are "ondisk". By this approach, the second issue can be solved.

System Design

As shown in Figure 1, the whole cross-cluster replication function involves some new components into RADOS. The first is the intermediate node which forwards repops to the backup cluster. The second is an "RP" module which stands for "replication" ---- please forgive me for this rough abbreviation, there isn't enough space in the figure. The "RP" module's work is to send repops to intermediate nodes.

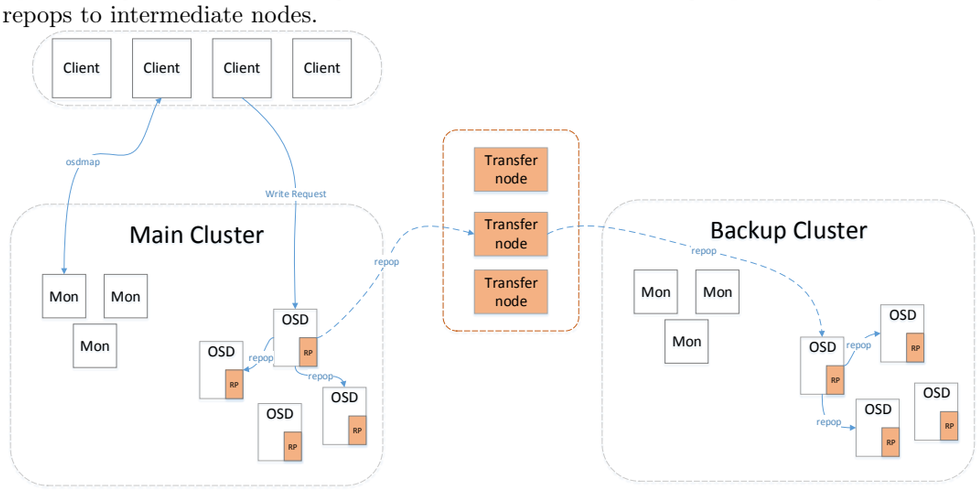

Figure 1 System Architecture

For the whole system to work, we need to add some more data structure or modify some existing data structure:

"RP" module

This module is mainly responsible for replicating repops to the intermediate nodes, it should contain a work queue *replication_wq* and a work thread *replication_thread*. Repops that need to be replicated are put into *replication_wq* by the OSD's work thread, and *replication_thread* continuously send those repops to intermediate nodes.

Modify class *MOSDOp* and *OpContext*

Add a bool type field *replicate* indicating whether the client requires operations targeting to the current object to be replicated. This field, like snap context, is used to give the client the authority to control whether their objects should be replicated.

Add *obj_set_id, obj_set_op_seq* and *obj_set_extents* field.

Modify class *RepGather*

Add a bool type field *replicated* indicating whether the repop has been replicated to the other cluster.

Add bool type field *replicate_sent*, true if the repop has been sent to intermediate nodes but has not been replicated.

Add long type field *replication_version*, a sequentially, monotonously increasing id to uniquely identify a repop within a single "up/acting set".

Add a bool type field *need_full_replicate* indicating whether the object needs a full-scale copy.

Add *obj_set_id, obj_set_op_seq* and *obj_set_extents* field.

Modify extensive attribute "OI"

Add a bool type field *replicate* indicating whether Ops to this object need to be replicated.

Add a bool type field *suspend* indicating whether Ops replication of this object is suspended.

Modify journal entry

Add a bool type mark indicating whether the current journal entry corresponds to a OP that needs to be replicated need to be added to the journal entry's metadata.

A pointer *replicated* needs to be added to the control structure of the OSD journal, pointing to the earlist repop that needs to be replicated and hasn't been replicated yet; as its counterpart, a pointer *replicate_head* pointing to the latest repop that needs to be replicated also needs to be added to OSD journal's control structure.

Modify pg_info_t

A long type field *first_replicating_version* indicating the oldest repop in *replication_wq's* version(not *RepGather::replicate_versi*
but pg log's version) needs to be added.

A bool type field *replicating* indicating whether the OSD was doing replication for Ops of the peering pg right before Peering needs also to be added.

These two fields are used to assist OP replication across a osd failure.

A queue *need_full_replicating* storing IDs of objects that needs a full copy needs to be added.

A long type field *obj_set_max_next_seq* indicating the max number of unreplicated object set op that the system can tolerate needs to be added.

Long type field *replication_version*, the same as *RepGather::replicate_version* needs to be added.

New interface

A new interface that let acting primary accept MOSDRepOp needs to be added for the backup cluster to be able to apply repops.

Besides these data structure modifications, some new op processing logic needs to be added or modified, I'll try to make these logic clear by using pseudocode:

Add replication logic to osd's client op process procedure

The basic idea is to let acting primary do the replicating, other replica osds just keep the repop in their *replication_wq*. In case of osd failure, other osds in the same up/acting set can continue the replication. When a repop is replicated, the acting primary is notified by intermediate nodes, then it has to notify replica osds of the completion. After the notification, both primary and replica osds mark that repop as "replicated", and removes continuous "replicated" repops at the tail of *replication_wq*. As the replication use the same journal space as the client i/o processing, there has to be a mechanism that prevent replication to use up the journal space. Here we provide an option *cutoff_threshold* to let user

determine the maximum space between *journal.replicated* and *journal.replicate_ head*. The "journal write" procedure needs also to be modified to let it only use the space outside the interval [*journal.replicated*, *journal.replicate_ head*]. When the *cutoff_ threshold* is met on acting primary or replica, replication should be suspended and When the *replicate_ start_ threshold* is met, replication should be restarted with a full object copy. The problem here is, as mentioned before, replication is done on acting primary, and it has to suspend when a replica's *cutoff_ threshold* is met, so there must be a way for acting primary to be ackownledged of the current replica's journal space usage. We did this by encapsulating replica's journal space usage in its "ondisk" message. The acting primary check this information every time it has to process a client op. The whole procedure is as described by Algorithm 1˜7.

Add additional logic to PEERING, RECOVERY and BACKFILL procedure

For the replication to be correct when encountering the osd failure, additional process logic needs to be added to the peering and recovery/backfill procedure. The basic idea is to make osds that needs to be recovered or backfilled select those osds who has the ability to continue the replication for the recovering object as recovery source, while the "object recovery source" replicate all repops related to the recovering object before start pushing. For this to work, first, we have to add replication information in the pg_info to let other osds know the progress of the replication going on locally. Then other osds use this information when they try to determine the recovery source of an object. As for now, we think that it should be appropriate for osds to select those osds that were replicating the recovering object right before the osd failure happens or whose *pg_ info.first_ replication_ version* is the largest as the recovery source, which should speed up the replication. In addition, when the recovery/backfill procedure completes, osds outside the pg's up/acting set should drop their repops that are targeting object in the pg when removing the pg. This is done by removing those repops upon the receiving of the MOSDPGRemove message. Other operations that need to be done are rebuilding the content of *replication_ wq* when doing journal_replay, and dropping repops that are received before the peering when the pg gose to "active + clean" since these repops should have been replicated by its recovery source. The whole process is as described by Algorithm 8˜14.

Another thing that has to be taken care of is the journaling process

Since both ordinary client op and replication use the osd's journal, there must be some mechanism to prevent them from interfere with each other. The basic idea is to make client op processing use the journal space outside [*journal.replicated, journal.replicate_ head*]. The detail is shown in Algorithm 15˜16.

Basic logic of the intermediate node

Intermediate nodes' main task is to accept replication messages sent by osds in master cluster, and send them to backup cluster. To preserve the cross-object consistency, intermediate nodes' must make sure that only when all repops in a obj set op arrived that they can go on sending them to the backup cluster. The detail is shown in Algorithm 17.

Algorithm 1 Primary OSD's repliation of I/O

Require:
op_ctx: op to be processed by acting set

```
1: long_before ← false;
2: replica_journal_too_full ← false;
3: replica_journal_restart ← true;
4: pg_info ← pgs[repop.repop_id.pgId].info
5: for rep_osd in reposd_set do
6:     if rep_osd.replication_version - pg_info.replication_version ≥ MAX_REP_INTERVAL_PRIM_REP then
7:         long_before ← true;
8:     end if
9:     if rep_osd.used_journal_space ≥ journal.cutoff_threshold then
10:        replica_journal_too_full ← true;
11:    end if
12:    if rep_osd.used_journal_space > journal.replicate_restart_threshold then
13:        replica_journal_restart ← false;
14:    end if
15: end for
16: if journal.meet_cutoff_threshold
    OR long_before OR replica_journal_too_full
    OR repop.obj_set_op_seq ≥ pg_info.obj_set_max_next_seq.find(repop.obj_set_id) then
17:    obj_info.suspend ← true;
18:    mark obj_info dirty;
19:    repop.need_replicate ← false;
20:    if journal.meet_cutoff_threshold OR long_before OR replica_journal_too_full then
21:        pg_info.need_full_replicate.blocked_by_journal_cutoff.insert(repop.repop_id.obj_id);
22:    else if repop.obj_set_op_seq ≥ pg_info.obj_set_max_next_seq.find(repop.obj_set_id) then
23:        blocking_seq ← pg_info.obj_set_max_next_seq.find(repop.obj_set_id);
24:        pg_info.need_full_replicate.blocked_by_obj_set_max.insert(repop, blocking_seq);
25:    end if
26: else if journal.meet_replicate_start_threshold
    AND !long_before AND replica_journal_restart
    AND repop.obj_set_op_seq ≤ pg_info.obj_set_max_next_seq - OBJ_SET_INTERVAL_SEQ then
27:    if obj_info.suspend AND op_ctx.replicate then
28:        obj_info.suspend ← false;
29:        mark obj_info dirty;
30:        repop.need_full_replicate ← true;
31:        repop.need_replicate ← true;
32:    end if
33: else
34:    if obj_info.suspend then
35:        repop.need_replicate ← false;
36:    end if
37: end if
38: if op_ctx.replicate AND !obj_info.replicate then
39:    obj_info.replicate ← true;
40:    mark obj_info dirty;
41: end if
42: if repop.need_replicate then
43:    repop.replication_version ← pg_info.replication_version.atomic_increment();
44: end if
45: repop ← new REPOP(op_ctx)
46: for rep_osd in reposd_set do
47:    repop.replica_osds.insert(rep_osd);
48:    rep_osd.issue(repop);
49: end for
50: write repop to journal
51: if repop.need_replicate then
52:    repliation_wq.need_replicate.push_back(repop);
53:    if is_primary(repop.repop_id.pgId) then
54:        replication_wq.pg_primary[repop.repop_id.pgId] ← true
55:    end if
56: end if
```

Algorithm 2 Upon any op is all_committed

Require:
repop: committed op

```
1: things needed to be done for repop all_committed
2: if repop.need_replicate then
3:     replication_wq.to_replicate.insert(repop)
4: end if
```

Algorithm 3 replication_thread

```
1: while replication_wq.to_replicate.haveNext() do
2:     replication_wq.to_replicate.peak_front(&repop)
3:     repop.replicate_sent ← true;
4:     replication_wq.to_replicate.pop()
5:     if repop.need_full_replicate then
6:         transfer_node.push_full_obj(repop.repop_id.obj_id);
7:     end if
8:     transfer_node.issue(repop);
9: end while
```

Algorithm 4 Upon any op is replicated

Require:
repop_id: the reply of the replicated repop

```
1: repop ← replication_wq.need_replicate.find(repop_id)
2: pg_info ← pgs[repop.repop_id.pgId].info
3: last_max_seq ← pg_info.obj_set_max_next_seq[repop.obj_set_id];
4: pg_info.obj_set_max_next_seq[repop.obj_set_id] ← repop.obj_set_op_seq + MAX_UMCOMMITTED_ALLOWED;
5: for obj_id in replication_wq.need_full_replicate.blocked_by_obj_set_max.find(repop.obj_set_id,
   repop.obj_set_max_block) do
6:     create full_replicate_repop for obj_id;
7:     if full_replicate_repop not in replication_wq then
8:         replication_wq.need_replicate.insert(full_replicate_repop);
9:         replication_wq.to_replicate.insert(full_replicate_repop);
10:    end if
11: end for
12: repop.replicated ← true
13: for iter in replicate_wq.need_replicate orderly do
14:    if !iter.replicated then
15:        if (journal.neet_cutoff_threshold AND pg_info.need_full_replicate.count(repop.repop_id.obj_id)) then
16:            break;
17:        end if
18:    end if
19: end for
20: for rep_osd in repop.replica_osds do
21:    rep_osd.notify_replicated(repop_id);
22: end for
23: move journal.replciated to iter.journal_seq
24: all_replica_meet_restart ← true;
25: for rep_osd in reposd_set do
26:    if rep_osd.used_journal_space > journal.replicate_restart_threshold then
27:        all_replica_meet_restart ← false;
28:    end if
29: end for
30: if all_replica_meet_restart AND journal.meet_replicate_start_threshold then
31:    for obj_id in pg_info.need_full_replicate.blocked_by_journal_cut_off do
32:        create full_replicate_repop for obj_id;
33:        if full_replicate_repop not in replication_wq then
34:            replication_wq.need_replicate.insert(full_replicate_repop);
35:            replication_wq.to_replicate.insert(full_replicate_repop);
36:        end if
37:    end for
38: end if
39: replication_wq.need_replicate.pop_until(iter);
```

Algorithm 5 Replica OSD receiving I/O operation

Require:
repop: op sent by acting primary

```
1: write repop to journal;
2: if repop.need_replicate then
3:     replication_wq.need_replicate.push_back(repop);
4:     reply_primary(repop.replication_version, used_journal_space);
5: end if
```

Algorithm 6 Upon receiving replica's I/O operation reply

Require:
repop_reply: op sent by acting primary

```
reposd_set[repop_reply.osd].used_journal_space ← repop_reply.used_journal_space;
all_replica_meet_restart ← true;
for rep_osd in reposd_set do
    if rep_osd.used_journal_space > journal.replicate_restart_threshold then
        all_replica_meet_restart ← false;
    end if
end for
if all_replica_meet_restart AND journal.meet_replicate_start_threshold then
    for obj_id in pg_info.need_full_replicate.blocked_by_journal_cut_off do
        create full_replicate_repop for obj_id;
        if full_replicate_repop not in replication_wq then
            replication_wq.need_replicate.insert(full_replicate_repop);
            replication_wq.to_replicate.insert(full_replicate_repop);
        end if
    end for
end if
```

Algorithm 7 Upon replica OSD notified of repop replicated

Require:
repop_id: id of the op that's replicated

```
pg_info ← pgs[repop.repop_id.pgId].info
repop ← repliation_wq.need_replicate.find(repop_id);
repop.replicated ← true;
for iter in replication_wq.need_replicate orderly do
    if !iter.replicated then
        if (journal.neet_cutoff_threshold AND pg_info.need_full_replicate.count(repop.repop_id.obj_id)) then
            break;
        end if
    end if
end for
move journal.replciated to iter.journal_seq
replication_wq.need_replicate.pop_until(iter);
```

Algorithm 8 Upon receiving pg_query_t::INFO

```
pg.info.first_replicating_version ← long.MAX_VALUE
for iter in replication_wq.need_replicate orderly do
    if iter.repop_id.pg_id ≤ pg.info.pg_id then
        pg.info.first_replicating_version ← iter.repop_id.version
        break;
    end if
end for
if replication_wq.pg_primary[pg.pgId] AND pg.info.first_replicating_version != long.MAX_VALUE then
    pg.info.replicating ← true;
end if
```

Algorithm 9 replicate all ops in build_push_op

Require:
push_obj: id of the object that's to be pushed

```
if progress.first then
    if obj_infos[push_obj].replicate AND !obj_infos[push_obj].suspend then
        for repop in replication_wq.need_replicate.find(push_obj) do
            if !repop.replicate_sent AND repop not in repop.replicate_wq.to_replicate then
                replication_wq.to_repliate.push_back(repop);
                repop.replicate_sent ← true;
            end if
        end for
        replication_wq.wait_for_complete(push_obj);
    end if
    do the rest...
end if
```

Algorithm 10 select recover source

Require:
obj_to_recover: id of the object that's to be recovered.

```
for iter in missing_loc.find(obj_to_recover) orderly do
    if peer_info[*iter].replicating then
        recover_source ← *iter
        break;
    end if
    if peer_info[*iter].first_replicating_version ≤ peer_info[recover_source].first_replicating_version then
        recover_source ← *iter
    end if
end for
```

Algorithm 11 clear divergent op in PGLog::merge_log

for *op* in *divergent* **do**
  iter ← *replication_wq.need_replicate*.find(*op.soid*, *op.version*);
  if *iter* != *replication_wq.need_replicate*.end() **then**
    if all repops before *iter* in *replication_wq.need_replicate* is replicated **then**
      journal.replicated ← *iter.journal_op*
    end if
  end if
  replication_wq.need_replicate.remove(*iter*);
end for

Algorithm 12 clear replication_wq when receiving MOSDPGRemove

Require:
  pg_list: pgs to remove
for *repop* in *replication_wq.need_replicate* **do**
  if *repop.repop_id.pgId* in *pg_list* **then**
    replication_wq.need_replicate.remove(*repop*);
  end if
end for
for *pgId* in *pg_list* **do**
  replication_wq.pg_primary.remove(*pgId*);
end for

Algorithm 13 rebuild replication_wq during journal replay

for *jentry* in *journal* **do**
  if *jentry.repop.need_replicate* **then**
    replication_wq.need_replicate.push_back(*jentry.repop*);
    replication_wq.pg_primary[*jentry.repop.repop_id.pgId*] ← *true*;
    pgs[*repop.repop_id.pgId*].*before_peering* ← *replication_wq.need_replicate.tail*
  end if
end for

Algorithm 14 drop repops befor *replication_wq.need_replicate.before_peering* when "going clean"

for *iter* in *replication_wq.need_replicate* orderly **do**
  if *iter* == *pg.before_peering* **then**
    break;
  end if
  if *iter.repop_id.pgId* == *pg.pgId* **then**
    replication_wq.need_replicate.remove(*iter*);
  end if
end for

Algorithm 15 determining *journal.meet_cutoff_threshold* and *journal.meet_replicate_start_threshold*

procedure SUBMIT_ENTRY
  if (*journal.write_pos* ≥ *journal.replicated*) **then**
    if *journal.write_pos* - *journal.replicated* ≥ *journal.cutoff_threshold* **then**
      journal.meet_cutoff_threshold ← *true*;
    end if
  else
    if *journal.replicated* - *journal.write_pos* ≥ *journal.cutoff_threshold* **then**
      journal.meet_cutoff_threshold ← *true*;
    end if
  end if
  do other normal things...
end procedure

procedure COMPLETE_THRU
  if (*journal.write_pos* ≥ *journal.replicated*) **then**
    if *journal.write_pos* - *journal.replicated* ≤ *journal.replicate_start_threshold* **then**
      journal.meet_replicate_start_threshold ← *true*;
    end if
  else
    if *journal.replicated* - *journal.write_pos* ≤ *journal.replicate_start_threshold* **then**
      journal.meet_replicate_start_threshold ← *true*;
    end if
  end if
  do other normal things...
end procedure

Algorithm 16 journal writes related operations

```
1: procedure SUBMIT_ENTRY
2:     if repop.need_replicate then journal.replicate_head ← journal.write_pos
3:     end if
4:     other normal work...
5: end procedure
6:
7: function WRITE_JOURNAL(op)
8:     if journal.write_pos + op.size > journal.replicated then
9:         if journal.header.start - journal.replicate_head > op.size then
10:            do_journal(journal.replicate_head, op);
11:            journal.write_pos ← journal.replicate_head + op.size;
12:        else
13:            return JOURNAL_FULL
14:        end if
15:    end if
16:    other normal work...
17: end function
```

Algorithm 17 intermediate node handle replication requests

```
1: procedure HANDLE_REPLICATION(rpp)
2:     oset ← obj_sets.find(rpp.obj_set_id);
3:     if rpp.obj_set_op_seq ≥ oset.last_committed_seq + MAX_UNCOMMITTED_ALLOWED then
4:         rpp.session.reply_error(TOO_MANY_UNCOMMITTED);
5:     end if
6:     oset_ops ← obj_set_ops.find_or_create(rpp.obj_set_op_seq);
7:     oset_ops.insert(rpp);
8:     if !oset_ops.obj_set_extents AND rpp.obj_set_extents then
9:         oset_ops.obj_set_extents ← rpp.obj_set_extents;
10:    end if
11:    oset_ops.obj_set_extents_received.merge(rpp);
12:    if oset_ops.obj_set_extents_received == oset_ops.obj_set_extents then
13:        oset_ops.all_received ← true;
14:    end if
15: end procedure
16:
17: procedure ISSUE_OS_OP_ENTRY
18:    for oset in obj_sets do
19:        iter ← oset.obj_set_ops.iterator();
20:        while iter.hasNext() do
21:            if iter→all_received then
22:                issue_op(*iter);
23:                oset.issued_ops.push_back(*iter);
24:                oset.obj_set_ops.remove(*iter);
25:            else
26:                break;
27:            end if
28:        end while
29:    end for
30: end procedure
31:
32: procedure ON_COMMIT(rpp_reply)
33:    oset ← obj_sets.find(rpp_reply.obj_set_id);
34:    leveldb.put(oset.obj_set_id, oset.last_committed_seq);
35:    for rpp′ in oset.issued_ops.find(rpp_reply.obj_set_op_seq) do
36:        rpp′→session.reply(rpp_reply);
37:        oset.issued_ops.remove(rpp′);
38:    end for
39: end procedure
```