High Level Design
MDT reply reconstruction improvement

version: 0.5
date: 2015/04/09
author: Grégoire Pichon - gregoire.pichon@bull.net

Introduction

Currently, the MDT cannot handle more than one single filesystem-modifying RPC at a time, because there is only one slot per client in the MDT last_rcvd file. This slot is used to save the state of the last transaction (transaction number, xid, RPC result, operation data) so the reply can be reconstructed in case of RPC resend if the reply was lost. As a consequence, the filesystem-modifying MDC requests are serialized, leading to poor metadata performance from a single Lustre client.

The goal of this project is to support multiple slots per client for reply reconstruction of filesystem-modifying MDT requests, and to improve metadata operations performance of a single client.

This work is tracked with Lustre JIRA LU-5319.

Solution Implementation

The design of the solution has been mostly inspired from an experimental patch developed by Alexey Zhuravlev.

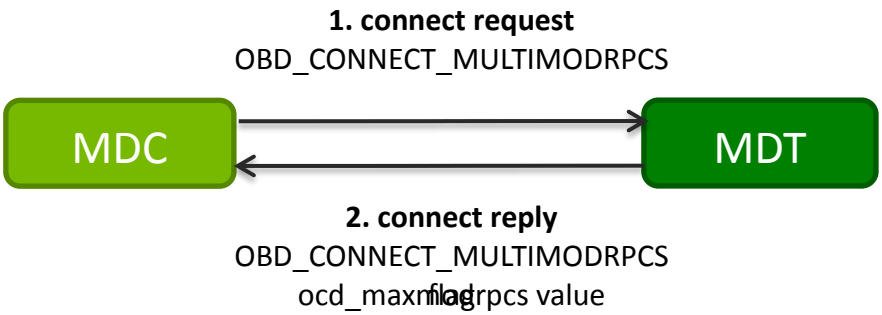
The support of multiple modify RPCs in flight has been primarily identified for metadata operations (MDC client and MDT target). This is the scope of the enhancement referenced by LU-5319 and of this document.

However, since the OST targets would also benefit from this enhancement (for lockless I/O or attr_set operations for example), the design has been made to facilitate reuse for any type of target. This explains why most of the server side modifications are made at target level rather than at MDT level.

Connection

The connection will support

- the new flag `ODB_CONNECT_MULTIMODRPCS` that indicates support of multiple filesystem-modifying RPCs in parallel.
- the new connection data field `ocd_maxmodrpcs` in the `obd_connect_data` structure that specifies the maximum number of modify RPCs per client supported by the server



The resulting connection flag and value will be available on the client in the *import* mdc procfs file.

```
# cat /proc/fs/lustre/mdc/fsms-MDT0000-mdc-ffff880836220c00/import
import:
    name: fsms-MDT0000-mdc-ffff880836220c00
    target: fsms-MDT0000_UUID
```

```
    state: FULL
    connect_flags: [ version, acl, xattr, inode_bit_locks, getattr_by_fid, no_oh_for_devices,
max_byte_per_rpc, early_lock_cancel, adaptive_timeouts, lru_resize, fid_is_enabled,
version_recovery, pools, large_ea, full20, layout_lock, 64bithash, jobstats, umask, einprogress,
lvb_type, flock_deadlock, disp_stripe, open_by_fid, lfsck, multi_mod_rpcs, dir_stripe ]
    connect_data:
       flags: 0x6784e79c344d1a0
       instance: 4
       target_version: 2.6.0.0
       max_brw_size: 1048576
       ibits_known: 0x3f
       grant_block_size: 0
       grant_inode_size: 0
       grant_extent_overhead: 0
       max_easize: 4012
       max_object_bytes: 0
       max_mod_rpcs: 8
...
```

Client side

The client will specify the OBD_CONNECT_MULTIMODRPCS flag within the connection request to the MDT.

The client obd structure will be extended to manage the number of modify RPCs in flight.

```
struct client_obd {
       ...
       /* modify rpcs in flight
        * currently used for metadata only */
       spinlock_t               cl_mod_rpcs_lock;
       __u16                    cl_max_mod_rpcs_in_flight;
       __u16                    cl_mod_rpcs_in_flight;
       __u16                    cl_close_rpcs_in_flight;
       wait_queue_head_t        cl_mod_rpcs_waitq;
       unsigned long           *cl_mod_rpcs_tag_bitmap;
       ...
}
```

The maximum number of modify RPCs in flight `cl_max_mod_rpcs_in_flight` will be initialized to `OBD_MAX_RIF_DEFAULT - 1` (=7). Two new routines will be provided to set or get this value from another Lustre component.

```
__u16 obd_get_max_mod_rpcs_in_flight(struct client_obd *cli);
int   obd_set_max_mod_rpcs_in_flight(struct client_obd *cli, int max);
```

The `cl_max_mod_rpcs_in_flight` value will be dynamically tunable through the `max_mod_rcps_in_flight` mdc procfs file, with the two following requirements:

- the value cannot exceed the maximum number of modify RPCs per client returned by the server during connection establishment
- the value cannot exceed or equal the maximum number of RPCs in flight (`max_rpcs_in_flight`)

The number of modify RPCs in flight and the number of close RPCs in flight will be accounted during RPC request preparation in order to enforce the maximum threshold. The `cl_mod_rpcs_in_flight` field will account the modify RPCs including the close RPCs. The `cl_close_rpcs_in_flight` will account the close RPCs. When the maximum is reached (`cl_mod_rpcs_in_flight >= cl_max_mod_rpcs_in_flight`), the thread preparing the request will be put to sleep on the client obd wait queue `cl_mod_rpcs_waitq`.

There will be an exception for close requests (MDS_CLOSE or MDS_DONE_WRITING). If the maximum is reached, but no other close request is in flight, the RPC will be sent anyway. This will prevent a potential deadlock where the processing of a modify metadata request triggers a lock cancellation, which requires a close request to be sent from the same client (see Lustre Bugzilla 3462). This explains why maximum modify RPCs in flight must be strictly lower than maximum RPCs in flight.

Two new routines will be used to acquire and release a modify RPC slot.

```
__u32 obd_get_mod_rpc_slot(struct client_obd *cli, __u32 opc, struct lookup_intent *it)
void obd_put_mod_rpc_slot(struct client_obd *cli, __u32 opc, struct lookup_intent *it, __u32
tag)
```

Every calls to `mdc_get_rpc_lock()` and `mdc_put_rpc_lock()` routines will be replaced by calls to `obd_get_mod_rpc_slot()` and `obd_put_mod_rpc_slot()` routines respectively. The `mdc_rpc_lock` structure will be removed.

It is necessary to manage additional information at client side to help the release of reply data in memory on the server (bitmap in client_obd structure for tag assignment, xid of the highest consecutive received reply in obd_import structure, ...). This is described in the section "Reply data release".

Server side

The OBD_CONNECT_MULTIMODRPCS flag will be added to the mask of connection flags supported by the MDT.

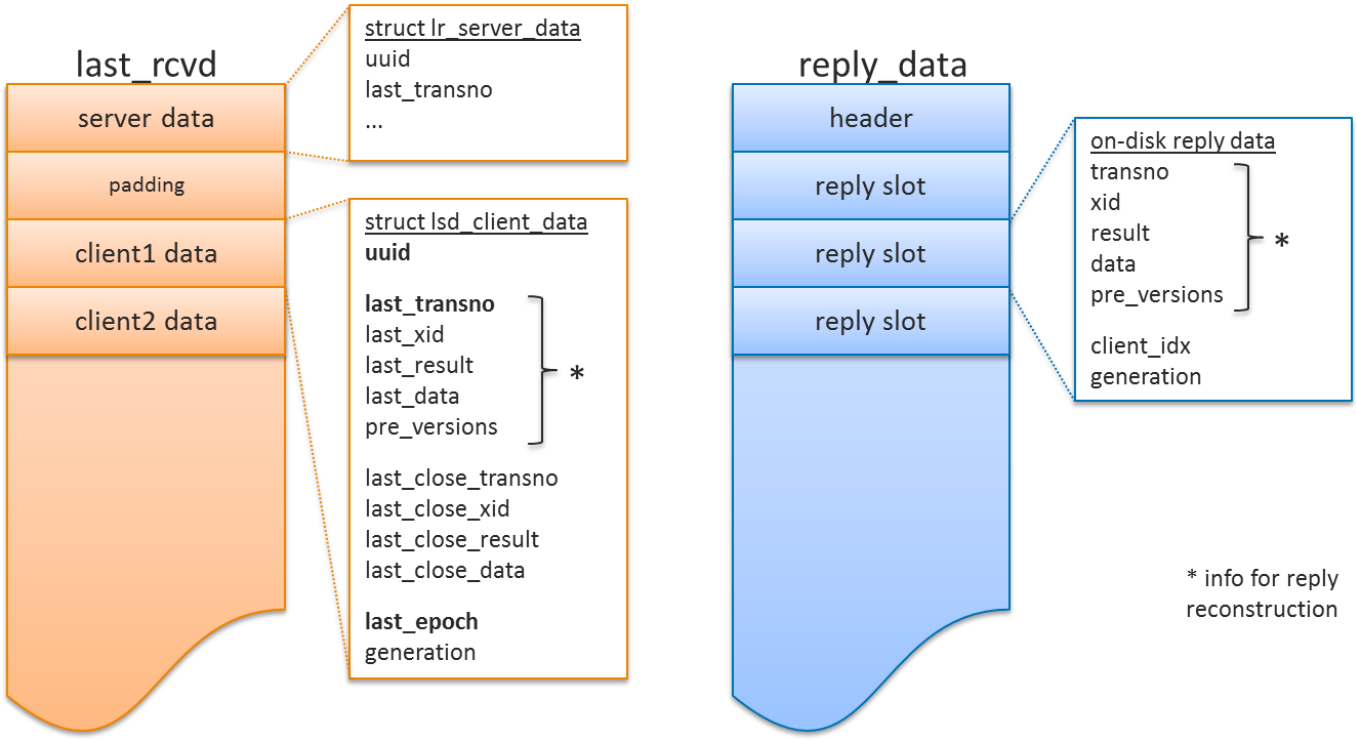
Tunable

A new parameter `max_mod_rpcs_per_client` will be added to the `mdt` kernel module to specify the maximum number of RPCs in flight allowed per client (8 by default). The parameter value will be returned to the MDC as part of the connection data (`ocd_maxmodrpc`) built in `mdt_connect_internal()` routine. It will be possible to dynamically update the parameter value but the change will be effective for new MDC connections only.

```
# modinfo mdt
filename:       /lib/modules/xxx/extra/kernel/fs/lustre/mdt.ko
license:        GPL
description:    Lustre Metadata Target (mdt)
...
parm:               max_mod_rpcs_per_client:maximum number of modify RPCs in flight per client
(uint)
```

On-disk data

The MDT target currently stores in the "last_rcvd" internal file both server and per-client information on the transactions that have completed. This allows to reconstruct the reply, in case reply was lost and a client resend its last request. The "last_rcvd" file starts with the `lr_server_data` structure and then contains one area for each connected client: `lsd_client_data` structure.



With the support of multiple modify RPCs in parallel, the "last_rcvd" file will no more store in the per-client area the transaction details used for reply reconstruction. A new MDT internal file, called "reply_data", will be created to store the reply data of every clients. It will be formatted with a header structure `lsd_reply_header` and many slots of structure `lsd_reply_data`. Any client will be able to use any free slot.

The "last_rcvd" file will still store the server data (`lr_server_data`) and some of the client data (fields `uuid`, `last_epoch`). Client data will be updated only when client connects or disconnects. The `first_epoch` field of `lsd_client_data` structure, which is currently not used, will be replaced by a slot generation count that will be incremented each time the slot is assigned to a new client. This is required at recovery for reply data restoration.

```
struct lsd_reply_data {
        __u64   lrd_transno;    /* transaction number */
        __u64   lrd_xid;        /* transmission id */
        __u64   lrd_data;       /* per-operation data */
        __u64   lrd_pre_versions[4]; /* versions for VBR */
        __u32   lrd_result;     /* request result */
        __u32   lrd_client_idx; /* client index in last_rcvd file */
        __u32   lrd_generation; /* client slot generation */
        __u32   lrd_pad[15];
};

struct lsd_reply_header {
        __u32   lrh_magic;
        __u32   lrh_header_size;
        __u32   lrh_reply_size;
        __u32   lrh_pad[sizeof(struct lsd_reply_data)-12];
};
```

Since intent disposition is 64-bits, the `lrd_data` field is defined as 64-bits.
The `lsd_reply_data` structure is padded to a power-of-two size, so that if fits into disk blocks evenly.
The `lsd_reply_head` structure is the same size as `lsd_reply_data` structure so that the structures are all aligned.

In-memory data

The `lu_target` structure will be extended to manage the "reply_data" file. It will record the `dt_object` structure to access the file and a bitmap to mark the reply data slots that are in use. The bitmap will be allocated on-demand by chunk of 1 million slots.

```
#define LUT_REPLY_SLOTS_PER_CHUNK  (1<<20)
#define LUT_REPLY_SLOTS_MAX_CHUNKS 16
struct lu_target {
```

```
        ...
        /** reply_data file */
        struct dt_object        *lut_reply_data;
        /** Bitmap of used slots in the reply data file */
        unsigned long           *lut_reply_bitmap[LUT_REPLY_SLOTS_MAX_CHUNKS];
};
```

The in-memory reply data will be represented by the `tg_reply_data` structure. The reply data of the modify RPCs sent by a client will be chained together and anchored in the `tg_export_data` structure.

```
struct tg_reply_data {
       struct list_head        trd_list;
       struct lsd_reply_data   trd_reply;
       int                     trd_index;
};

struct tg_export_data {
       ...
       /** List of reply data */
       struct list_head        ted_reply_list;
       int                     ted_reply_cnt;
       /** Reply data with highest transno is retained */
       struct tg_reply_data    *ted_last_reply;
};
```

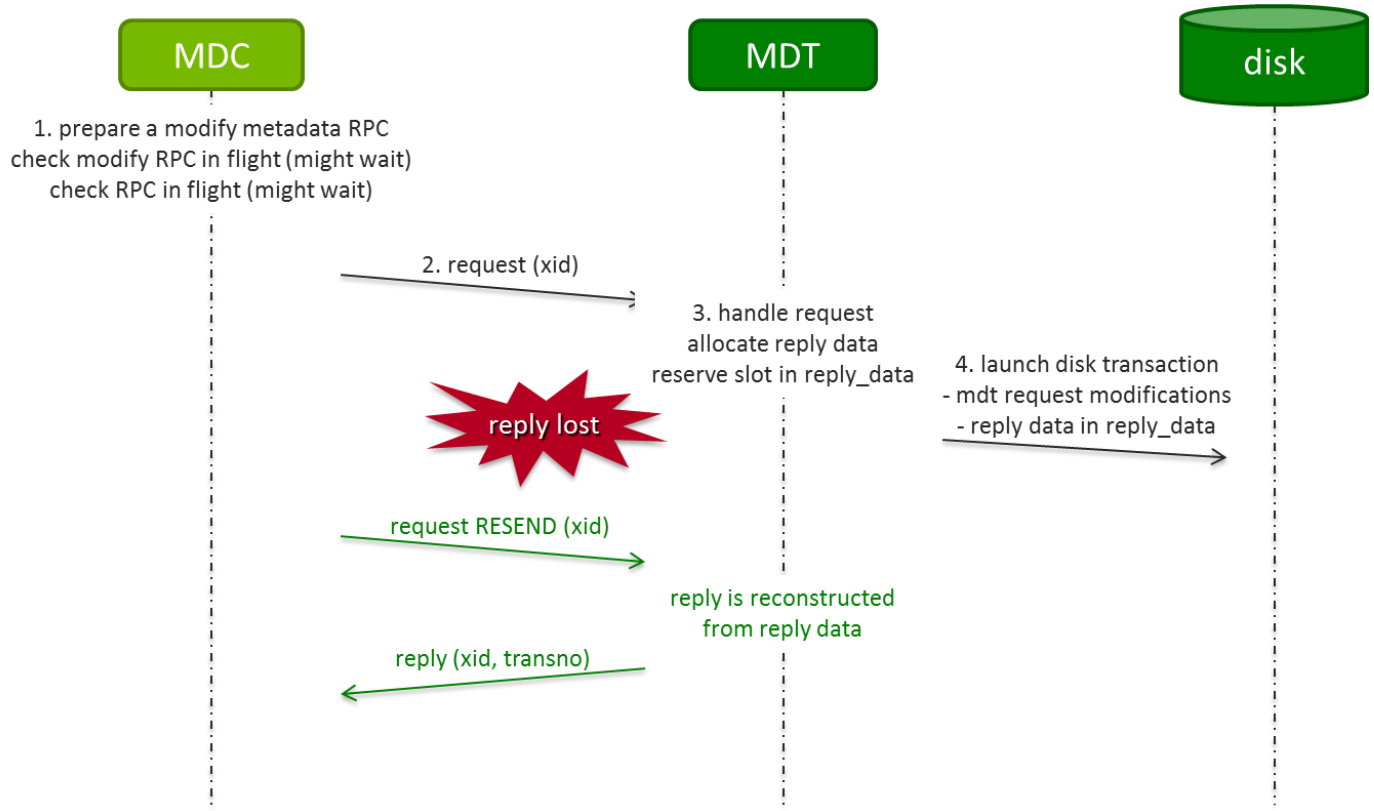
At the end of server RPC handling, a free slot will be selected from the reply bitmap and a reply data structure will be allocated and filled. The disk transaction will include both the modifications related to the RPC operation and the write of reply data in the selected slot of "reply_data" file.

The release of in-memory reply data structure and of the slot in the bitmap will be done after the client received the RPC reply. This is detailed in the section "Reply data release".

The reply data with highest transaction number will be retained. This means the slot in the "reply_data" file will not be released. This will ensure the export's highest committed transno can be rebuilt from disk in case of recovery.

RPC resend

When the MDT will receive an RPC with the MSG_RESENT or MSG_REPLAY flag, it will look for a matching xid in the reply list of the corresponding target export and rebuilt the reply, as described in the following figure.

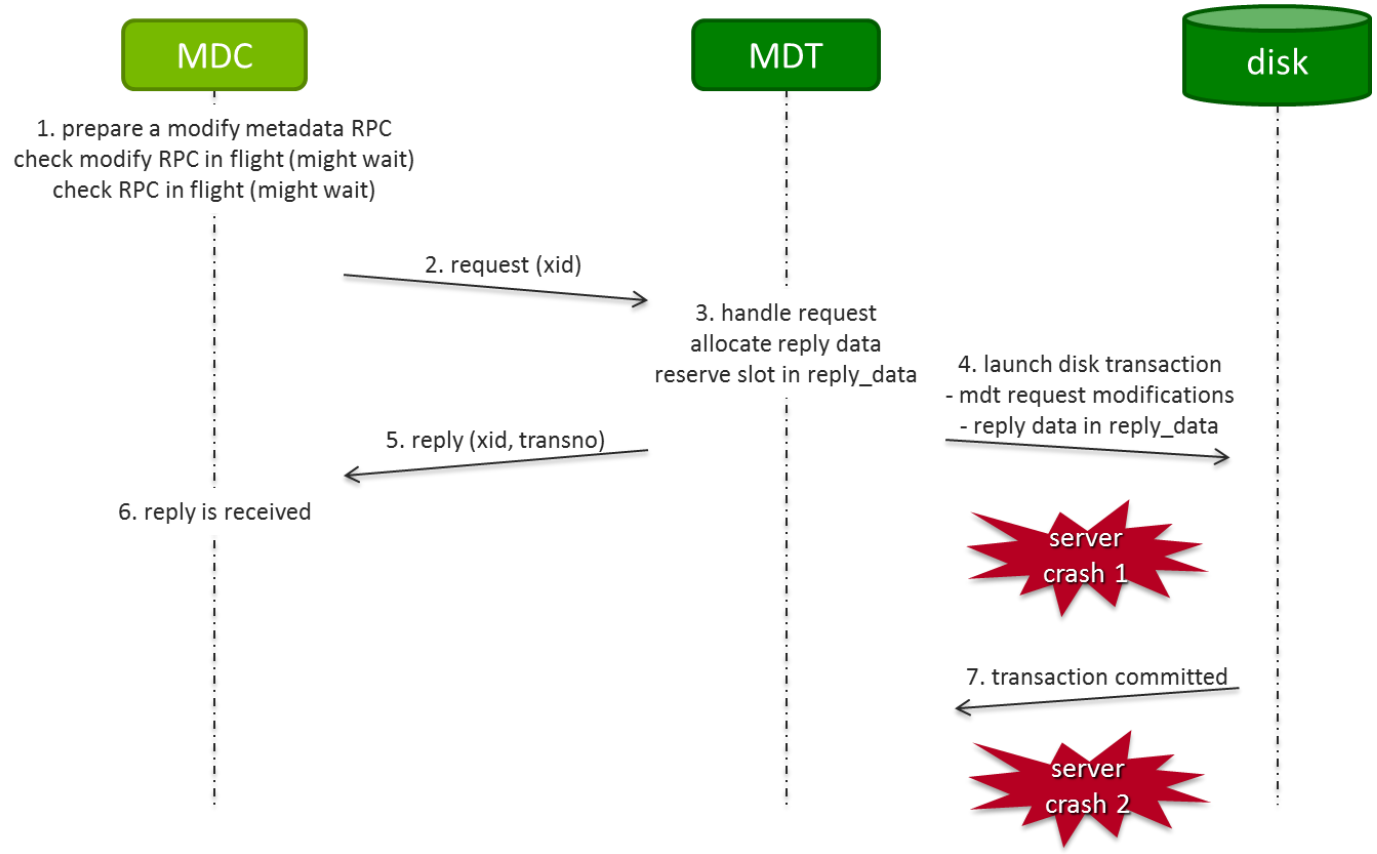


Target recovery

In case of target recovery, the "last_rcvd" file will continue to be used to restore target information from server data, and to restore exports information from per-client data. Additionally, the "reply_data" file will be used to restore the in-memory reply data associated to each export. The `lrd_client_idx` and `lrd_generation` field of on-disk reply data will allow to uniquely identify the export the reply data belong to.

Note that since on-disk reply data will only be cleared by overwriting new reply data, and reply slots will not be assigned to a specific client, it will be possible to have a lot of reply slots containing old reply data for a client. It means a lot of old reply data will be restored in memory at target recovery.

During the recovery, the replay of the RPC requests will be handled as described by the next figure.
If the server crashes between step 4 and step 7, nothing was written to disk neither the mdt request modification, nor the reply data. Therefore the request is normally handled by the MDT.
If the server crashes after step 7 and client replay the RPC request, reply data is restored to memory from reply_data file and the request is handled with reply reconstruction.



Reply data release

Three different alternatives have been studied to manage the release of the in-memory reply data and the freeing of reply bitmap slot.

1. tag reuse

This solution was the one implemented in Alexey's prototype. It consists in adding a tag value within each RPC. The value will be set by the client when preparing the RPC. The value will be 0 for non-modify RPCs and between 1 and the maximum allowed modify RPC in flight for the modify RPCs. The client will maintain a bitmap of in-use tag values and ensure each tag value is set to only one RPC in flight at a time.
Using the tag information embedded in each new RPC, the server will be able to release the reply data having the same tag than the new RPC for the corresponding target export. Reuse of a tag assumes client received the reply of the previous RPC with the same tag.

The drawback of this solution is that the server keeps in memory several reply data for each client, even after RPC replies have been received by the clients. It represents max allowed modify RPC in flight times the number of clients.

2. reply acknowledgment

With this solution the reply data will be released when the server will receive the reply-ack, giving confirmation that the client received the reply. Since not every kind of RPC request the acknowledgment of the reply (close RPC for instance), it will be necessary to force reply-ack for every requests that leaded to the creation of a reply data.

The drawback of this solution is that the reply-ack message might be lost and is not resent in that case. It also does not allow to release the deprecated reply data that are restored at target recovery.

3. received xid

This solution consists in embedding in every RPC "the highest xid for which a reply has been received and does not have an unreplied lower-numbered xid". The client will maintain the value per device within the `obd_import` structure and update it each time a reply is received. To facilitate the processing, the list of sent RPCs (`imp_send_list`) will be ordered by xid. Another possible implementation will be to parse the list of sent RPCs (`imp_send_list`) and the list of delayed RPCs (`imp_delayed_list`) and look for the lowest unreplied xid, each time a new RPC is sent.

Using that highest xid embedded in each new RPC, the server will be able to release the reply data having a xid lower or equal to the one specified in the new RPC.
Since the `pb_last_xid` field of the `ptlrpc_body` structure is currently unused, the solution will be implemented with that field.
Additionally, when the target recovers, this solution allows to release all the old reply data that have been restored when the first RPC arrives to the server.

The drawback of this solution is that if an RPC takes a long time to be replied (due to network issue or long server processing for instance), other RPCs with higher xids will not have their reply data released. This would potentially make "reply_data" file significantly grow.

Considering the pros and cons of each alternatives, the 1st solution based on tag reuse and the 3rd solution based on the received xid will be used simultaneously.

Client disconnection

When a client disconnects, the server will release all the in-memory reply data for that client. When all the clients are disconnected, the "reply_data" file will be truncated.

API

Management of slots in "reply_data" internal file

```
static int tgt_find_free_reply_slot(struct lu_target *lut);
static void tgt_set_reply_slot(struct lu_target *lut, int idx);
static void tgt_clear_reply_slot(struct lu_target *lut, int idx);
```

Management of the content of the "reply_data" internal file (on-disk reply_data)

```
static int tgt_reply_data_write(const struct lu_env *env, struct lu_target *tgt, struct
lsd_reply_data *lrd, loff_t off, struct thandle *th);
static int tgt_reply_data_read(const struct lu_env *env, struct lu_target *tgt, struct
lsd_reply_data *lrd, loff_t off);
static int tgt_reply_header_write(const struct lu_env *env, struct lu_target *tgt, struct
lsd_reply_header *lrh);
static int tgt_reply_header_read(const struct lu_env *env, struct lu_target *tgt, struct
lsd_reply_header *lrh);
```

Initialization of the "reply_data" internal file or restoration of its content at target recovery

```
int tgt_reply_data_init(const struct lu_env *env, struct lu_target *tgt);
```

Management of in-memory reply data

```
static void tgt_free_reply_data(struct lu_target *lut, struct tg_export_data *ted, struct
tg_reply_data *trd);
static void tgt_release_reply_data(struct lu_target *lut, struct tg_export_data *ted, struct
tg_reply_data *trd);
struct lsd_reply_data *tgt_lookup_reply(struct ptlrpc_request *req);
int tgt_handle_tag(struct ptlrpc_request *req);
int tgt_handle_received_xid(struct obd_export *exp, __u64 rcvd_xid);
```

Interoperability

When a client that supports multiple modify RPCs in flight will connect to a server that does not support the feature, the reply will not have the `OBD_CONNECT_MULTIMODRPCS` flag and therefore the `cl_max_mod_rpcs_in_flight` value will be set to 1. This will ensures the client don't send more than one modify RPC, or one close RPC, or one non-close and one close RPC at a time.

When a client that does not support multiple modify RPCs in flight will connect to a server that does support the feature, the server will store the reply data information in the "last_rcvd" file, as before. This is because, without the client received xid information, the server would not be able to release the reply data in memory.

Lustre Upgrade / Downgrade

The upgrade of a client from a version that does not support multiple modify RPCs in flight to a version that does support the feature, or the downgrade of a client from a version that does support multiple modify RPCs in flight to a version that does not support the feature will work without any change, since the client behavior is not related to any persistent information.

When upgrading a server from a version that does not support multiple modify RPCs in flight to a version that does support the feature, the content of the "last_rcvd" file will be restored as before (server and client area restoration code will not been modified) and a new "reply_data" file will be created.

Downgrading a server from a version that does support multiple modify RPCs in flight to a version that does not support the feature will be allowed only if there was no more valid reply data on disk (ie. no client connected) at the time the server stopped or crashed. This can be managed with an incompatibility flag in the `server_compat_data structure tgt_scd[LDD_F_SV_TYPE_MDT]`. This flag will prevent old code to mount target. It will be set on disk in `lsd_feature_incompat` field of `lr_server_data` structure as soon as a client connects, and cleared on disk once all the clients are disconnected.