Interoperable Client Recovery

Amit Sharma

28-12-2007

1 Introduction

This document outlines the detailed design for the Interoperable client recover work which was explained in the HLD (interop-client-recovery.lyx)

2 Functional Specification

2.1 Working in interoperability mode

- At the time of connection with the server the client would use the OBD_CONNECT_FID flag to find if the server is a 1.8 server. If the server is a 1.8 server, the client can use this information for all deciding if somethings are to be done differently.
- (req->rq_export->exp_connect_flags & OBD_CONNECT_FID) and (exp->exp_connect_flags & OBD_CONNECT_FID), can be used to find if the client is connected to a 1.8 server.

2.2 Two cases of recovery

- When the server has to be upgraded, it would be brought down and then brought up after the upgrade. So, the client when connecting to the upgraded server will see this as a case of recovery and has to handle it.
- The other case of recovery is the normal case, when either the client or the server or the network goes down and then comes up again.

2.3 Algorithm Change

- As discussed in the HLD there has been a slight change in the recovery algorithm from the point of view of the client. This change has to be taken care of in the target_handle_connect() on the client side.
- Just to get an idea of the algorithm change and for the sake of reference, below is the section from HLD which discusses about it.

1. In 1.8 server the recovery task is handled by the recovery thread target_recovery_thread, which is started off by target_recovery_init(). This is different from 1.6 in which the recovery process is started as part of the target_handle_connect.
2. At the server end when the recovery process starts, a recovery_timer is started and the first transno to be replayed is found. And the recovery_timer is stopped after the time out period. If some client fails to connect within this time, it is evicted.
3. In the next stage, the clients replay their requests. If some client fails to replay its requests in the time out period it is evicted.
4. The next stage is for the clients to replay their locks. But if any client is unable to replay the locks in the time out period, the recovery is aborted and all clients are evicted. And the recovery process has to start again.
5. After this the server drops the recovering flag, and starts forwarding all the requests from now on to the regular mds_handle().
6. Then in the final stage, the server sends out final reply.

- Most of the above is similar to 1.6, except that there are just two stages in 1.6 Replay and Final Reply. This will have some minor affect in the recovery process for the client in 1.6.x. And has to be taken care of by enabling the 1.6.x client to communicate with the 1.8 server in a way that the 1.8 client would behave in a similar situation.
- We discuss below for how we propose to deal with the above difference in 1.6 and 1.8 client.

2.4 The other differences between 1.6 and 1.8

Apart from the above, there were a few more differences between 1.6 and 1.8 that were mentioned as part of the HLD which might come into play. But most of them are being taken care of as part of other tasks and very little if any work related to those differences would come in the purview of this task. For the sake of reference we mention those differences here again and also for the sake of need in case some corner cases which are recovery specific arise out of those differences which might have to be handled as part of this task.

- Use of FIDs - Handled as part of interop_client_fid
- Request flag and connection flag changes - Handled as part of client_interop_reqs and mixed_layout_reqs
- SOM related changes - Handled as part of som-recovery

3 Use Cases

3.1 Server upgrade/downgrade

- The server upgrade/downgrade will be treated as a case of recovery from the point of view of the client. While upgrading the server to 1.8, it will be failed and for the clients it will be a normal recovery procedure. Just that in this case, they would be talking to a server of higher or lower version (depending on upgrade/downgrade) after the recovery.

- To handle this scenario, a flag would be needed which would tell us what the previous version of the server was. So, after every recovery a comparison to this flag would tell us if there has been a upgrade/downgrade at the server.

- In case there has been a upgrade/downgrade at the server, the client would need to take care of converting the import data that it has to the new server type. But this would depend on the way the task client_interop_fid and client_interop_reqs implement some of the data structures which store most of the connection and server information.

3.2 Server crashes/reboots

- This case will be handled almost in the same way as it is done now. Just some small changes will have to be made to accommodate for the change in the algorithm that 1.8 uses (See section 2.3).

3.3 Client crashes/reboots

- Similar to the above case, this will also be handled in more or less the same way as it is done now. With small changes to adapt to the new recovery algorithm.

3.4 Network failure

- This case also remains mostly the same. The HLD has a good explanation of all the possible scenario that can come up in the case of a Network failure and what would be the action taken. Just for the sake of reference it is pasted below again.

  Network failure has to be handled in a similar way as the case when client crash/reboots. When the client requests time out, the recovery procedure will be kicked off and then the process follows as per the recovery algorithm

  We can take a look at how a “open-write-close” scenario would work in case there is a network failure.

  The network failure can happen at the following stages:

- Before the client does a "open": There could be two cases in open. One is the case of a pure open, and the other is the case of open/creat.
  - In case of pure open, the task is simple as a new transaction is not created in this case, and the trans no is just bumped up. In case of failure such a transaction can be taken care of at the server end by a simple replay of the open operation.
  - In the open/create case there will be a transaction and so it will have to be handled. In the transaction stop callback the transaction number, request id, last operation result and intent disposition is stored in the last_rcvd record. At the time of replay this record will come in handy and since the file had already been created, it would just be opened as part of the recovery.
- After the "open" but before "write": There could again be two cases here, one is the simple case where in the clients request for open has been executed by the server so after the reconnect nothing needs to be done at the server end. The other case would be when the server did not receive the open request from the client. So, based on the status of the request on the client, it would start to replay requests, the server would compare the requests and see if that request has been executed at the server. And decide on whether to replay the transaction or not.
- Network failure after the "write" but before "close" and failure after "close" : The file open request (along with the fid for the newly created file) will be kept in the client replay list until the file is closed. There is an open file handle (struct mdt_file_data *mfd) on MDS for every open file, linked together into this client's export. When client crash/reboot/reconnect to MDS, all open handle will be destroyed. When server crash/reboot/recover, client will replay its open request, and continue on the write operation.

4 Logic Specification

4.1 Working in interoperability mode

- At the time of connection with the server the client would use the OBD_CONNECT_FID flag to find if the server supports fids, if the server does support fids then it has to be a 1.8 server. If the server is a 1.8 server, the client can use this information for all deciding if somethings are to be done differently.
- For different places in the code where it may be needed to find what mode the client is running in (normal mode or interoperability mode). The exp_connect_flags and OBD_CONNECT_FID can be used as follows to find the mode, depending on the context of the call.

```
(req->rq_export->exp_connect_flags & OBD_CONNECT_FID)
```

```
or
(exp->exp_connect_flags & OBD_CONNECT_FID)
```

- And at other places in the code, couple of flags defined later in this document (server_version = PRE_FID and POST_FID) can also be used to determine which logic to implement.

```
if (server_version == PRE_FID) {
...
... //set of things to be done
}
or
if (server_version == POST_FID) {
...
... //another set of things to be done
}
```

4.2 Two cases of recovery

1. When the server has to be upgraded/downgraded, it would be brought down and then brought up after the upgrade/downgrade. So, the client when connecting to the upgraded/downgraded server will see this as a case of recovery and has to handle it.

   (a) For this to be implemented, the proposal is to have a store the information of the previous server connection. So, after every recovery the new server information (version number?) can be compared to the old information to see if the server has been upgraded. As of now we dont save the old server version number information. It can be saved without much trouble and used as below.

   In ptlrpc_connect_interpret( ), the previous connect flag can be compared to the current connect flag be made to come to a conclusion if the server has been upgraded.

```
ptlrpc_connect_interpret()
{
...
...
  if (!(ocd->ocd_connect_flags & OBD_CONNECT_FID)) {
        current_server_version = PRE_FID; // could be other name, this is
  } else {                                 // just for understanding.
        current_server_version = POST_FID;
  if (prev_server_version < current_server_version) {
        server_upgrade = true; // will be used later
  } elseif (prev_server_version > current_server_version) {
        server_downgrade = true;
  }
```

```
          prev_server_version = current_server_version;
                                  // store the current version as the
                                  // old version for future checks.
        ...
        }
```

(a) If there has been an upgrade at the server, depending on the implementation the import information might need updation to match the new server version and be compatible. That will have to be taken care at this stage. This is a case which is not very clear as of now and will depend on if the import information will change in the interoperability mode and so not much details are presented. There may be no need to do handle this situation in case there are no changes made to the import information.

```
if (server_upgrade) {
        // do a set of things to ensure that the old imports/data
        // are compatible with the new upgraded server;
        // this will depend on the implementation of how the
        // import/data is stored differently from the 1.6 in the
        // interoperability mode, hence the details are missing.
} elseif (server_downgrade) {
        // do a set of things to ensure that the old imports/data
        // are compatible with the down graded server;
        // this will depend on the implementation of how the
        // import/data is stored differently from the 1.6 in the
        // interoperability mode, hence the details are missing.
}
```

2. The other case of recovery is the normal case, when either the client or the server or the network goes down and then comes up again. This will be handled as a normal case and will follow the usual recovery algorithm, ofcourse following the modified algorithm which 1.8 uses.

4.3 Algorithm change

The change in recovery algorithm will affect the three use cases (3.2, 3.3 and 3.4) mentioned above. We see some details below as to how the changes can be handled. Not much work will be needed to handle the algorithm change mentioned. A quick comparison of the 1.6 and 1.8 code for the import recovery state machine (ptlrpc_import_recovery_state_machine()) shows that not many changes have been made to the client side code. So, for getting the 1.6.x (interoperable client) to talk to the 1.8 server very minor changes are needed.

```
diff 1.6/import.c 1.8/import.c [function = int signal_completed_replay() ]
896c889,890
```

```
<         lustre_msg_add_flags(req->rq_reqmsg, MSG_LAST_REPLAY);
---
>         lustre_msg_add_flags(req->rq_reqmsg,
>                              MSG_LOCK_REPLAY_DONE | MSG_REQ_REPLAY_DONE);
956c950
OR
It can be better understood as below :
if (server_version == PRE_FID) {
         lustre_msg_add_flags(req->rq_reqmsg, MSG_LAST_REPLAY);
} elseif (server_version == POST_FID) {
         lustre_msg_add_flags(req->rq_reqmsg,
                              MSG_LOCK_REPLAY_DONE | MSG_REQ_REPLAY_DONE);
}
```

The extra stage that we earlier mentioned is being taken care at the client side in a single step as seen in the above code diff. So it would be a similar case to emulate the above code for the new client aswell.

4.4 The other differences between 1.6 and 1.8

As previously mentioned most of these are being taken care of by other tasks. But in case they are unable to take care of some corner cases or very special cases which are very specific to recovery we can take up those tasks later.